Intel® C/C++ Compiler User's Guide

With Support for the Streaming SIMD Extensions

Issued in U.S.A.
Order Number 718195-001

Intel® C/C++ Compiler User's Guide: With Support for the Streaming SIMD Extensions

Order Number: 718195-001

Contents

Appendix A Compiler Limits

Appendix B Experimental Performance Tuning

Index

Examples

Figures

Tables

How to Use This Online Manual

Click to hide or show subtopics when the bookmarks are shown.

Double-click to jump to a topic when the bookmarks are shown.

Click to display bookmarks.

Click to display thumbnails.

Click to close bookmark or thumbnail view.

Click and use on the page to drag the page in vertical direction.

Click and drag on the page to magnify the view.

Click and drag on the page to reduce the view.

Click and drag the selection cursor to the page.

Click to go to the first page of the manual.

Click to go to the previous page.

Click to go to the next page.

Click to go to the last page.

Click to return to the previous view. Use this button when you need to go back after using the jump button (see below).

Click to go forward from the previous view.

Click to set 100% of the page view.

Click to display the entire page within the window.

Click to fill the width of the window.

Click to open a dialog to search for a word or multiple words.

Click jump button on manual pages to jump to the related subjects. Use the return back icon above to go back.

Printing an Online File: Select **Print** from the **File** menu to print an online file. The dialog that opens allows you to print full text, range of pages, or selection.

Viewing Multiple Online Manuals: Select **Open** from the **File** menu, and open the .PDF file you need. Select **Cascade** from the **Window** menu to view multiple files.

Resizing the Bookmark Area: Drag the double-headed arrow that appears on the area's border as you pass over it.

Jumping to Topics: In this manual, many of the function names and section titles are printed in green color, underlined style to indicate that you can jump to them.

On the next page, you will find a jump text demo.

Jump Text Demo

The following text segment is taken from the manual. In this example, the section name “Restricting Optimization” is displayed in green underlined font. If you click this section name, the first page of the section will display. To return to this page

click the  icon in the tool bar.

> This chapter describes ways to improve the performance of your application and the effects of optimization options on programs. The “Restricting Optimization with -Od” section tells you how to suspend optimizations for certain applications or for debugging.

About This Manual

This manual describes how to use the Intel® C/C++ Compiler for Win32* systems. The Intel C/C++ Compiler can be hosted on either a Windows NT* or Windows* 95 operating system.

- Chapters 1 through 3 help you get started.
- The "Compiler Option Quick Guide" in Chapter 2 alphabetically lists and briefly describes compiler options. It also provides cross-reference links to more detailed descriptions in this manual.
- Chapter 4 describes well-known performance optimizations.
- Chapter 5 describes Interprocedural and Profile-Guided Optimizations.
- Chapters 6 through 9 help you use preprocessing and maintain language conformance.
- Chapter 10 describes diagnostic information.
- Chapter 11 describes libraries.
- Chapter 12 describes how to control compiler-generated code.
- Chapter 13 describes support for MMX™ Technology and the Streaming SIMD Extensions.
- Chapter 14 describes vectorization options.
- Appendix A provides a table of the compiler limits.
- Appendix B provides information on experimental performance tuning.
- This manual also provides a glossary and an index.

This manual does not teach you the C programming languages. As prerequisites, you should be familiar with software development of C/C++ and assembly-language programs. You should also be familiar with the host computer's operating system and architecture of Intel® processors.

The Intel C/C++ Compiler operates similarly to the Microsoft Visual C++* compiler. In many cases, you can apply the functional descriptions provided with the Microsoft Visual C++ compiler to the Intel compiler, except where this manual states that you cannot. See "Application Development" in Chapter 1 for the relationship of the Intel C/C++ Compiler to system-specific programming support tools.

Related Publications

The following documents provide additional information relevant to the Intel C/C++ Compiler:

- *The Annotated C++ Reference Manual*, 3rd edition, Ellis, Margaret; Stroustrup, Bjarne, Addison Wesley, 1991. Provides information on the C++ programming language.
- *The C Programming Language*, 2nd edition, Kernighan, Brian W.; Ritchie, Dennis W., Prentice Hall, 1988. Provides information on the K & R definition of the C language.
- *C: A Reference Manual*, 3rd edition, Harbison, Samual P.; Steele, Guy L., Prentice Hall, 1991. Provides information on the ANSI standard and extensions of the C language.
- For environment specifics, see the manuals or on-line help supplied with Microsoft Visual C++, 32-bit edition for Windows.
- For Win32 specific information, see the documentation included with the *Microsoft Win32 Software Development Kit, Version 3.1*.

Information about the target architecture is available from Intel and from most technical bookstores. Some helpful titles are:

- *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture,* Intel Corporation, order number 243190
- *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual,* Intel Corporation, order number 243190
- *Intel Processor Identification with the CPUID Instruction,* Intel Corporation, order number 241618
- *Intel Architecture MMX Technology Programmer's Reference Manual,* Intel Corporation, order number 241618.
- *Pentium® Pro Processor Developer's Manual (3 Volume Set)*. Intel Corporation, order number 242693.

- *Pentium II Processor Developer's Manual*. Intel Corporation, order number 243502-001.
- *Pentium Processor Specification Update*. Intel Corporation, order number 242480.
- *Pentium Processor Family Developer's Manual*, Intel Corporation, order numbers 241428-005.
- Most Intel documents are also available from the Intel Corporation web site at *www.intel.com*.

Notational Conventions

This manual uses the following conventions:

`This type style`	Indicates an element of syntax, a reserved word, a keyword, a filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
	`l` indicates lowercase letter L in examples. `1` is the number 1 in examples. `O` is the uppercase O in examples. `0` is the number 0 in examples.
`This type style`	Indicates the exact characters you type as input.
`This type style`	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
`[`*`items`*`]`	Indicates that the items enclosed in brackets are options.
`{`*`item`*` | `*`item`*`}`	Indicates to elect only one of the items listed between braces. A vertical bar (|) separates the items.
`...` (ellipses)	Indicate that you can repeat the preceding item.

Overview

1

This chapter introduces you to the Intel C/C++ Compiler for Win32 Systems. The compiler is designed to operate on any 32-bit Intel Architecture machine. However, it produces optimized code for the Intel486™, Pentium®, Pentium Pro, Pentium II, Celeron™, Pentium II Xeon™, and Pentium III processors.

Tools You Need

To compile programs with this compiler you need the following tools:

- Windows NT, Windows 95, or Windows 98 operating systems
- Microsoft Visual C++ Versions 4.2 or higher available on the host operating system

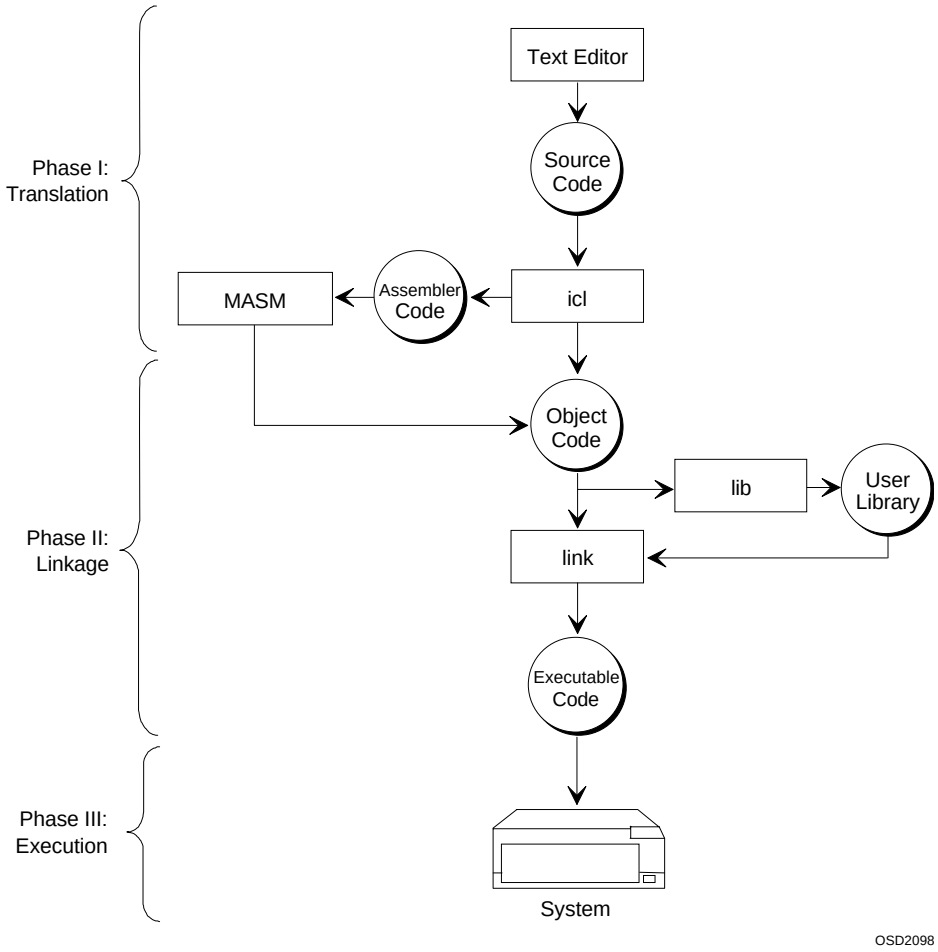
NOTE. *The Intel C/C++ Compiler is a plug-in product that uses the Microsoft Visual C++ header files, libraries, linker, and many other components.*

If you plan to assemble files generated by the assembler, you must provide an assembler. Version 6.12 of the Microsoft Macro Assembler (MASM) is recommended, but you can use Version 6.11 if you are not going to use the Streaming SIMD Extensions. Whatever assembler you use, you must name it `ML.EXE` and place it in your default path.

Application Development

Figure 1-1 shows you how the Intel compiler (`icl`) fits into the Microsoft application development environment. The compiler processes C or C++ language source and generates either assembly or object modules. You decide the input and output by setting options when you run the compiler.

Figure 1-1 Application Development Cycle



Compiler Operation 2

This chapter describes how to run the compiler, how to select compilation options, and the default behavior of the compiler.

This chapter also contains the "Compiler Option Quick Guide", which lists all the Intel C/C++ Compiler options. The options that you invoke change the compiler's default behavior in the following areas:

- controlling compilation flow
- controlling compiler output
- preprocessing files
- controlling optimization
- specifying language conformance
- preparing for debugging
- controlling the linker
- managing special cases
- preparing for profiling

The options available for the Intel C/C++ Compiler are described in detail in the remaining chapters of this manual.

Compiler Command-line Syntax

To invoke the compiler from the DOS command line, enter the following command:

```
prompt> icl [options] [path]filenames
```

options	Indicates one or more command-line options. The compiler recognizes one or more letters preceded by a hyphen (-) or a forward slash (/) as an option.
filenames	Indicates one or more source files to be processed by the compilation system. You can specify more than one [*path*]*filename*. You must separate multiple filenames with a space.

Using the Intel® C/C++ Compiler within the Microsoft Visual C++ Integrated Development Environment (IDE)

If you have Microsoft Visual C++ versions 4.2 or higher on your system, the installation procedure automatically plugs the Intel C/C++ Compiler into the Microsoft IDE. This provides compatibility between the Intel and Microsoft C/C++ compiler switch options. Additionally, you are enabled with Intel's optimizations, which are designed specifically to exploit the power of Intel architectures. To invoke the Intel C/C++ Compiler from the Microsoft IDE, follow these steps:

1. Click on Select Compiler from the Tools Menu.
2. The Microsoft Developer Studio displays the Select Compiler window, which allows you to choose a compiler.
3. To choose the Intel Compiler, click the check-box and specify the version in the down-arrow menu.
4. Click the OK button to return to the Microsoft IDE.

NOTE. *To switch back to the Microsoft compiler, clear the box for the Intel C/C++ Compiler.*

You can now use Microsoft IDE to build your application or portions of your application using the selected compiler. For more information on using the Intel C/C++ Compiler, click the Help button from the Intel Compiler Selection Tool in the Visual C++ development environment.

Filename Extensions

By, default the compiler recognizes `.cc`, `.cpp`, and `.cxx` files as C++ files. In examples, this guide uses the `.cpp` extension for C++ files. The compiler recognizes files with the `.c` extension as C files. Also, the Intel C/C++ Compiler recognizes the default filename extensions listed in Table 2-1.

Table 2-1 Default Filename Extensions

Filename	Interpretation	Action
filename.lib	object library	Passed to LINK.exe.
filename.i	C or C++ source preprocessed and expanded by the C/C++ preprocessor	Passed to compiler.
filename.obj	compiled object module	Passed to LINK.exe.
filename.asm	assembly file	Assembled by MASM.

If you use specify **`-TP`** *`[files]`*, the compiler treats all files on the command line as C/C++ files. Both `x.i` and `y.c` are treated as C++ files on the following command line:

```
prompt> icl -c -TP x.i y.c
```

However, if you specify **`-Tp`** *`file`*, the compiler treats the next file on the command line as a C++ file regardless of the extension. For example, to treat `x.i` as a C++ file, but `y.c` as a C file, you could type the following:

```
prompt> icl -c y.c -Tp x.i
```

NOTE. *Usually, you cannot associate a filename with an option by command-line placement alone. Specified options apply to all files. For example, in the following command line, the* `-Za` *option applies to both* `x.cpp` *and* `y.cpp`*:*

```
prompt> icl -c x.cpp -Za y.cpp
```

The `-Tc` *and* `-Tp` *are exceptions because they affect only the file that follows them on the command line.*

Compiler Option Quick Guide

Table 2-2 provides you with a reference of all compilation control options and some options for control of the linker.

Table 2-2 Summary of Command-line Options

Option	Description	Default	Reference
`-?, -help`	Prints compiler options summary.	OFF	page 10-1
`-c`	Stops the compilation process after an object file has been generated. The compiler generates an object file for each C or C++ source file or preprocessed source file.	OFF	page 6-4
`-C`	Places comments in preprocessed source output.	OFF	page 7-2
`-Dname[=value]`	Defines a macro name and associates it with the specified value.	OFF	page 7-3
`-E`	Stops the compilation process after the C or C++ source files have been preprocessed, and writes the results to `stdout`.	OFF	page 7-2
`-EHa`	Enables asynchronous C++ exception handling model.	OFF	*
`-EHc`	Specifies that `extern` C functions do not throw exceptions.	ON	*
`-EHs`	Enables the synchronous C++ exception handling model.	OFF	*
`-EP`	Stops the compilation process after the C or C++ source files have been preprocessed and writes the results to `stdout`. The `#line` directives are stripped (that is, not included in the output).	OFF	page 7-2
`-F n`	Sets the amount of stack to reserve for the program by passing `-stack:`*n* to the linker.	OFF	*

* The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-FA[c s]`	Produces an assembly output file. The `s` argument has no effect on the Intel C Compiler.	OFF	*
`-Fa[filename]`	Produces an assembly output file with the specified filename, or the default name if the filename is not specified.	OFF	page 6-5
`-Fe[filename]`	Produces an executable output file with the specified filename, or the default name if the filename is not specified.	ON	page 6-5
`-Fm[filename]`	Instructs the linker to produce a map file.	OFF	*
`-Fo[filename]`	Produces an object output file with the specified filename, or the default name if the filename is not specified.	ON	page 6-5
`-G4`	Targets the optimizations to the Intel i486 processor.	OFF	page 4-3
`-G5`	Targets the optimizations to the Intel Pentium processor.	OFF	page 4-3
`-G6`	Targets the optimizations to the Intel Pentium Pro and Pentium II processors.	ON	page 4-3
`-GA`	Optimizes for Microsoft Windows applications.	OFF	*
`-Ge`	Enables stack-checking.	ON	*
`-Gf`	Enables string-pooling optimization.	ON	*
`-GF`	Enables read-only string-pooling.	ON	*
`-Gh`	Adds a call to user-supplied `__penter` routine in each function prolog.	OFF	*
`-GR`	Enables C++ Runtime Type Information (RTTI).	OFF	*

* The options marked with the asterisk in the “Reference” column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-Gr`	Selects Microsoft `_fastcall` as the default calling convention.	OFF	*
`-GR-`	Disables C++ RTTI.	ON	*
`-Gs` *n*	Enables stack-checking for routines with *n* or more bytes of local variables and compiler temporaries.	OFF	*
`-GT`	Enables fiber-safe thread local storage (TLS).	OFF	*
`-GX`	Enables C++ exception handling.	OFF	*
`-GX-`	Disables C++ exception handling.	ON	*
`-Gy`	Enables reordering of functions at link time.	ON	*
`-H` *n*	Limits the length of external symbol names to *n* characters.	OFF	*
`-Idirectory`	Specifies an additional directory to search for include files.	OFF	page 3-4
`-J`	Makes the default `char` type unsigned.	OFF	*
`-LD`	Produces a DLL.	OFF	*
`-LDd`	Creates a debug version of DLL.	OFF	*
`-link`	Passes options to the linker.	OFF	page 3-5
`-MD`	Compiles and links with the dynamic, multi-thread C runtime library.	OFF	*
`-MDd`	Compiles and links with the dynamic, multi-thread, debug version of the C runtime library.	OFF	*
`-ML`	Compiles and links with the static, single-thread C runtime library.	ON	*
`-MLd`	Compiles and links with the static, single-thread, debug version of the C runtime library.	OFF	*

* The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
-MT	Compiles and links with the static, multi-thread C runtime library.	OFF	*
-MTd	Compiles and links with the static, multi-thread, debug version of the C runtime library.	OFF	*
-O1	Optimize for speed, but disable some optimizations that increase code size for small speed benefit. The -O1 option has the same effect as specifying the options: -Og, -Oi-, -Os, -Oy, -Ob1, -Gf, -Gs, and -Gy.	OFF	page 4-1
-O2	Optimizes for speed. The -O2 option has the same effect as specifying the following options:-Og, -Oi, -Ot, -Oy, -Ob1, -Gf, -Gs, and -Gy.	ON	page 4-1
-Ob*n*	Controls the compiler's inline expansion. The amount of inline expansion performed varies with the value of *<n>* as follows. 0 Disables inlining. 1 Enables inlining of functions declared with the _inline keyword. Also enables inlining according to the C++ language. 2 Enables inlining of any function. However, the compiler decides which functions to inline. Enables interprocedural optimizations and has the same effect as -Qip.	OFF	page 5-4
-Od	Disables optimizations.	OFF	page 4-1
-Og	Enables global optimizations.	ON	*
-Oi	Enables inline expansion of standard library functions.	ON	page 4-6

* The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
-Oi-	Disables inline expansion of standard library functions.	OFF	page 4-6
-Op	Favors conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic. Behavior for NaN comparisons does not conform.	OFF	page 4-7
-Op-	Favors optimization over conformance to the ANSI C and IEEE 754 standards for floating-point arithmetic.	ON	page 4-7
-Os	Enables most speed optimizations, but disable optimizations that increase code size for a small speed benefit.	OFF	*
-Ot	Enables all speed optimizations.	ON	*
-Ox	Same as the -O2 option without the -Gf and -Gy options.	OFF	*
-Oy	Enables the use of the ebp register in optimizations.	ON	page 6-6
-Oy-	Disables the use of the ebp register in optimizations. Instead, it is used as the frame pointer.	OFF	page 6-6
-P	Stops the compilation process after C or C++ source files have been preprocessed and writes the results to files named according to the compiler's default file-naming conventions.	OFF	page 7-2
-QA-	Causes all predefined macros (other than those beginning with __) and all assertions to be inactive. (Same as -u.)	OFF	page 7-3

* The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-QAname[(values)]`	Associates a symbol name with the specified sequence of values. Equivalent to an `#assert` preprocessing directive.	OFF	page 7-3
`-Qax[extensions]`	Generates specialized and generic code for processor-specific extensions.	OFF	page 4-4
`-QH`	Produces a list of include file dependencies, one per line, to `stdout`.	OFF	page 7-5
`-QI0f`	Avoids the incorrect decoding of certain 0f instructions for code targeted at older processors.	OFF	page 11-4
`-QIfdiv`	Enables a software patch for the floating-point division flaw that exists in some steppings of the Pentium processor.	OFF	page 11-3
`-QIfdiv-`	Disables the software patch for the floating-point division flaw that exists in some steppings of the Pentium processor.	ON	page 11-3
`-Qip`	Enables interprocedural optimizations.	OFF	page 5-2
`-Qip_no_inlining`	Disables inlining that would result from the `-Qip` or `-Ob2` interprocedural optimizations, but has no effect on other interprocedural optimizations. This option has no effect on user-directed inlining (`-Ob1`).	OFF	page 5-4
`-Qipo`	Enables interprocedural optimization across files.	OFF	page 5-2
`-Qkscalar`	Performs all 32-bit floating point arithmetic using the Streaming SIMD Extensions instead of the default x87 instructions.	OFF	page 14-3

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-Qlocation,tool,path`	Specifies an alternate version of a *tool* specified by *path*.	OFF	page 3-4
`-Qlong_double`	Changes the default size of the `long double` data type from 64 to 80 bits.	OFF	page 4-10
`-QM`	Generates makefile dependency lines for each source file, based on the `#include` lines found in the source file.	OFF	page 7-6
`-Qnobss_init`	Places variables that are initialized with zeroes in the DATA section.	OFF	page 12-2
`-Qoption,tool,list`	Passes an argument list to another program in the compilation sequence, such as the assembler or linker.	OFF	page 3-4
`-Qpc nn`	Enables floating-point significand precision control. The value of *nn* is used to round the significand to the correct number of bits. The value of *nn* must be either 32, 64, or 80.	nn=64	page 4-8
`-Qpf[options]`	Improves cache usage with prefetching.	OFF	page 4-5
`-Qprec`	Improves floating-point precision but does not provide the speed benefit of `-Op`.	OFF	page 4-8
`-Qprec_div`	Disables the floating point division-to-multiplication optimization.	OFF	page 4-8
`-Qprof_dir dirname`	Specifies the directory to hold profile information.	OFF	page 5-5
`-Qprof_gen`	Instruments the program to collect basic block execution counts.	OFF	page 5-5
`-Qprof_genx`	Generates an instrumented object file and also creates a new static profile information file (`.spi`).	OFF	page 5-5
`-Qprof_use`	Uses dynamic feedback information.	OFF	page 5-5

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-Qrcd`	Disables changing of the FPU rounding control.	OFF	page 4-10
`-Qrestrict`	Enables pointer disambiguation with the `restrict` qualifier.	OFF	page 14-3
`-Qsfalign-`	Disables stack alignment for all functions.	OFF	page 13-44
`-Qsfalign16`	Aligns stack for functions with 16 byte variables.	OFF	page 13-44
`-Qsfalign8`	Aligns stack for functions with 8 or 16 byte variables.	ON	page 13-44
`-Qunroll`*n*	Specifies the maximum number of times to unroll a loop.	*n* = 0	page 4-6
`-Quse_asm`	Compiles source files to assembly files and calls MASM to generate object files automatically.	OFF	page 6-4
`-Qvc4`	Enables compatibility with Visual C++ 4.x.	OFF	page 9-2
`-Qvc5`	Enables compatibility with Visual C++ 5.0.	ON	page 9-2
`-Qvc6`	Enables compatibility with Visual C++ 6.0.	ON	page 9-2
`-Qvec`	Enables the vectorizer.	OFF	page 14-2
`-Qvec_alignment`	Controls the default alignment of vectorizable data.	OFF	page 14-2
`-Qvec_emms[-]`	Controls whether the compiler automatically inserts an EMMS instruction after vectorization of loops.	OFF	page 14-3
`-Qvec_no_alias[-]`	Assumes that no aliasing can occur between objects with different names.	OFF	page 14-4
`-Qvec_no_arg_alias[-]`	Assumes on entry that procedure arguments are not aliased.	OFF	page 14-4
`-Qvec_verbose`	Controls the diagnostic levels produced by the vectorizer.	OFF	page 14-3

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-Qwd`*`tag`*	Disables the soft diagnostic that corresponds to *`tag`*.	OFF	page 10-5
`-Qwe`*`[tag]`*	Overrides the severity of the soft diagnostic corresponding to *`tag`* and makes it an error.	OFF	page 10-5
`-Qwn`*`num`*	Limits the number of errors displayed prior to aborting compilation to *`num`*.	100	page 10-6
`-Qwr`*`[tag]`*	Overrides the severity of the soft diagnostic corresponding to *`tag`* and makes it a remark.	OFF	page 10-5
`-Qww`*`[tag]`*	Overrides the severity of the soft diagnostic corresponding to *`tag`* and makes it a warning.	OFF	page 10-5
`-Qx[extensions]`	Generates specialized code for processor-specific extensions.	OFF	page 4-4
`-S`	Stops the compilation process after an assembly source has been generated. The compiler writes assembly code source for each C or C++ source file or preprocessed source file to a file using the output file's naming conventions.	OFF	page 6-2
`-TC`	Compiles all source or unrecognized file types as C source files.	OFF	*
`-Tc` *`file`*	Treats *`file`* as a C source file.	OFF	*
`-TP`	Compiles all source or unrecognized file types as C++ source files.	OFF	*
`-Tp` *`file`*	Treats *`file`* as a C++ source file.	OFF	*
`-u`	Disables all predefined macros (other than those starting with __) and assertions. (Same as -QA-.)	OFF	*

* The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-Uname`	Suppresses any definition of a macro name; equivalent to a `#undef` preprocessing directive.	OFF	page 7-3
`-V text`	Sets version string to *text*.	OFF	*
`-vd0`	Suppresses C++ hidden `vtordisp` constructor/destructor displacement member.	OFF	*
`-vd1`	Generates C++ hidden `vtordisp` constructor/destructor displacement member.	ON	*
`-vmb`	Selects the smallest representation for pointers to members. Use this option if you define each class before you declare a pointer to a member of the class.	ON	*
`-vmg`	Selects the general representation for pointers to members. Use this option if you declare a pointer to a member before you define the corresponding class.	OFF	*
`-vmm`	Enables pointers to class members with single or multiple inheritance with `-vmg`.	OFF	*
`-vms`	Enables pointers to members of single-inheritance classes with `-vmg`.	OFF	*
`-vmv`	Enables pointers class members of any inheritance type with `-vmg`.	OFF	*
`-W0`	Specifies to only display error messages (no warnings or remarks).	OFF	page 10-4
`-W1, -W2, -W3`	Specifies that error and warning messages are displayed.	ON	page 10-5

* The options marked with the asterisk in the “Reference” column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Table 2-2 Summary of Command-line Options (Continued)

Option	Description	Default	Reference
`-W4`	Specifies that error, warning, and remark messages are displayed.	OFF	page 10-6
`-X`	Removes the standard directories from the list of directories to be searched for include files.	OFF	page 3-3
`-Za`	Enforces strict conformance to the ANSI standard for C.	OFF	page 8-2
`-Zd`	Produces only line number information in the object file (for debugging).	OFF	*
`-Ze`	Accepts extended C language.	ON	page 8-2
`-Zi`	Generates symbolic debugging information in the object code for use by source-level debuggers.	OFF	page 6-6
`-Zl`	Disables embedding default libraries in object files.	OFF	*
`-Zp[`*`number`*`]`	Specifies the strictest alignment constraint for structure and union types as one of the following: 1, 2, 4, 8, or 16 bytes.	8	page 12-1
`-Zs`	Checks the syntax of a program and stops the compilation process after the C or C++ source files and preprocessed source files have been parsed. Generates no code and produces no output files. Warnings and messages appear on `stderr`.	OFF	page 6-2

* The options marked with the asterisk in the "Reference" column produce the same results for the Intel C/C++ Compiler as for the Microsoft Visual C++ Compiler. If you need more information on these options, see the documentation that accompanies Microsoft Visual C++.

Default Behavior of the Compiler

You need not enter any options when you invoke the Intel C/C++ compiler. If you do not specify any options, the compiler uses default settings. The compiler driver performs the following actions by default:

- Produces executable output with the filename of the first source or object file on the command line with a `.exe` suffix.
- Optimizes for the Pentium Pro or Pentium II processors with `-G6`.
- Searches for include files using the `INCLUDE` variable.
- Searches for library files in directories specified by the `LIB` variable, if they are not found in the current directory.
- Sets 16 bytes as the strictest alignment constraint for structures.
- Displays error and warning messages.
- Uses ANSI with extensions (`-Ze`) for C source files.
- Performs standard optimizations using the default `-O2` option, as described in "Optimization Choices" in Chapter 4.

If the compiler does not recognize a command-line option, that option is ignored and a warning is displayed. See, Chapter 10, Diagnostic Information for detailed descriptions about system messages.

Changing the Compilation Environment 3

This chapter describes how to customize the environment you use during compilation. Through customizing you can specify the following:

- names and locations of compilation tools
- options to include during each compilation
- options and files to use for individual projects
- directories in which to search for include files and libraries

Environment Variables

To customize your environment, you can specify paths where the compiler can search for special files such as libraries or include files. On Win32* systems, you can specify these variables in the Environment tab of the System Properties window. You can access this window by clicking on the System icon in the Control Panel. The following variables are relevant to your compilation environment:

- `LIB` specifies the directory path for the math libraries
  Also, the compiler calls `LINK.exe`, the Microsoft linker, to produce an executable file from the object files. This linker searches the path specified in the `LIB` environment variable to find the libraries.
- `PATH` specifies the directory path for the compiler executable files
- `INCLUDE` specifies the directory path for the for include files.
- `TMP` specifies the directory in which to store temporary files. If the directory specified by `TMP` does not exist, the compiler places the temporary files in the current directory.

Configuration Files

A user configuration file named `icl.cfg` can be added to the directory where `icl.exe` is installed. In this file, you can specify common options that you want to include in each compilation. Options in the configuration file are processed in the order they appear, followed by the command-line options that you specify when you invoke the compiler.

You can decrease the time you spend entering command-line options and ensure consistency by using the configuration file to automate some command-line entries. However, you should be aware that options placed in the configuration file will be included each time you run the compiler. If you have varying option requirements for different projects, see "Response Files" later in this chapter.

You can insert any valid command-line option into this file. Use the pound "`#`" character to treat the rest of the line as a comment.

Example 3-1 illustrates a sample `icl.cfg` file.

Example 3-1 Sample icl.cfg File

```
## Sample icl.cfg file.
##
## Define preprocessor macro MY_PROJECT.
-DMY_PROJECT
##
## Additional directories to be searched for include
## files, before the default.
-Ic:\project\include
##
## Use the static, multi-threaded C run-time library.
-MT
```

Response Files

You can use response files to specify options used during particular compilations and save this information in individual files. Response files are invoked as an option on the command line. Options in a response file are inserted in the command line at the point where the response file is invoked.

You can decrease the time you spend entering command line options and ensure consistency by using response files to automate command line entries. Further, you can use individual response files to maintain options for specific projects and avoid editing the configuration file when changing from one project to the next.

Any number of options or filenames can be placed on a line in the response file. You can also use several response files in the same command line. You can insert any valid command-line option into this file. Use the pound "#" character to treat the rest of the line as a comment.

The syntax for using response files is as follows:

```
prompt> icl @response_file filenames
```

Note that an "at" sign (@) must precede the name of the response file on the command line.

Include Files

By default, the compiler searches for the standard include files in the directories specified in the INCLUDE environment variable. You can also indicate the location of include files in the configuration file, icl.cfg, using the options described in "Specifying an Include Directory (-I)".

Specifying an Include Directory (-I)

Use the -I*directory* option to specify an additional directory in which to search for include files. Included files are brought into the program with a #include preprocessor directive. The compiler searches directories for include files in the following order:

- directory of the source file that contains the include
- directories specified by the -I option
- directories specified in the INCLUDE environment variable

Removing Include Directories (-X)

Use the -X option to prevent the compiler from searching the default path specified by the INCLUDE environment variable.

You can use the -X option with the -I option to prevent the compiler from searching the default path for include files and direct it to use an alternate path. For example, to direct the compiler to search the path \alt\include instead of the default path, do the following:

```
prompt> icl -X -I \alt\include newmain.cpp
```

How To Specify Alternate Tools and Paths

The Intel compiler lets you go outside the default paths and tools to specify alternate tools for preprocessing, compilation, assembly, and linking. Further, you can invoke options specific to your alternate tools on the command line. This functionality is provided by -Qlocation and -Qoption.

Specifying an Alternate Component (-Qlocation,*tool*,*path*)

Use -Qlocation to specify an alternate path for a tool. This option accepts two arguments using the following syntax:

```
prompt> -Qlocation,tool,path
```

tool Indicates one of the following to designate the compilation tool to receive one or more of these arguments:

cpp	Specifies the compiler front-end preprocessor.
c	Specifies the C/C++ compiler.
asm	Specifies the assembler.
link	Specifies the linker.

Passing Options to Other Programs (-Qoption, *tool*, *optlist*)

Use -Qoption to pass an option specified by *optlist* to a *tool*, where *optlist* is a comma-separated list of options. The syntax for this command is the following;

```
Qoption,tool,optlist
```

tool	Indicates one of the following that designate the compilation tool to receive one or more of these arguments:
	`cpp` Specifies the compiler front-end preprocessor.
	`c` Specifies the C/C++ compiler.
	`asm` Specifies the assembler.
	`link` Specifies the linker.
optlist	Indicates one or more valid argument strings for the designated program. If the argument is a command-line option, you must include the hyphen. If the argument contains a space or tab character, you must enclose the entire argument in quotation characters (`" "`). You must separate multiple arguments with commas.

The following example directs the linker to create a memory map when the compiler produces the executable file from the source.

```
prompt> icl -Qoption,link,-map:proto.map proto.cpp
```

The `-Qoption,link` option in the preceding example is passing the **`-map`** option to the linker. This is an explicit way to pass arguments to other tools in the compilation process.

To get the same results implicitly you could also use the following:

```
prompt> icl -Fmproto.map proto.cpp
```

Passing Options to the Linker (-link)

To pass options specifically to the linker, use the `-link` option. For example, to compile the file `a.cpp` and instruct the linker to link `a.obj` with `libfoo.lib`, creating `a.exe`, type the following at the prompt:

```
prompt> icl a.cpp -link libfoo.lib
```

NOTE. *The compiler passes everything following* `-link` *to the linker only. Therefore, all other compiler options must precede* `-link`.

Optimizations

4

This chapter describes ways to improve the performance of your application by using the optimization options on programs. You can find related information on optimizations in Chapter 5, Interprocedural and Profile Guided Optimizations and Appendix B, "Experimental Performance Tuning," which describes some experimental ways to tune your code.

Optimization Choices

To specify the kind of optimization you want for your program, use the following command-line options:

`-Od`	Disables optimizations.
`-O1`	Enables options `-Og`, `-Oi-`, `-Os`, `-Oy`, `-Ob1`, `-Gf`, `-Gs`, and `-Gy`. However, `-O1` disables a few options that increase code size for a small speed benefit such as inline expansion of library functions. This option is not as aggressive as the Microsoft Visual C++ Compiler version of the `-O1` option. In most cases, `-O2` is recommended over `-O1` because the `-O2` option enables inline expansion, which helps programs that have many function calls.
`-O2`	Enables options `-Og`, `-Oi`, `-Ot`, `-0y`, `-Ob1`, `-Gf`, `-Gs`, and `-Gy`. Confines optimizations to the procedural level. See Table 4-1 for a comprehensive list of optimizations. The `-O2` option is on by default.

Restricting Optimization with -Od

The `-Od` option disables optimizations. Table 4-1 shows the optimizations that the compiler applies when you invoke the optimization in the **Option** column. The only optimization not disabled by `-Od` is the one that has "any" in the **Option** column.

Table 4-1 Optimization Summary

Optimization	Affected Aspect of Program	Option
constant propagation	constants and expression evaluation	-O1 / -O2
copy propagation	constants and expression evaluation	-O1 / -O2
dead-code elimination	instruction sequencing	-O1 / -O2
global register allocation	register use	-O1 / -O2
instruction scheduling	instruction reordering	-O1 / -O2
loop unrolling	instruction sequencing	-O1 / -O2
loop-invariant code movement	instruction sequencing	-O1 / -O2
optimized code selection	instruction selection/ addressing modes	any
partial redundancy elimination	constants and expression evaluation	-O1 / -O2
strength reduction/induction variable simplification	instruction selection/sequencing constants and expression evaluation	-O1 / -O2
variable renaming	register use	-O1 / -O2

The Effect of -Od on -Oy and Debugging

When you specify `-Od`, you automatically disable `-Oy`, which is otherwise enabled by default or when you specify `-O1` or `-O2`. The `-Oy` option allows the compiler to use the `ebp` register as a general purpose register in optimizations. However, most debuggers expect `ebp` to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-Oy-` option instructs the compiler to generate code that maintains and uses `ebp` as a stack frame pointer, without turning off other `-O1` or `-O2` optimizations. As a result, the debugger can still produce a stack backtrace. However, using `-Oy-` reduces the number of available general-purpose registers by one, and can result in slightly less efficient code.

Targeting a Processor (-G*n*)

Use -G*n* to target an application to run on a specific processor for maximum performance. Regardless of which of the -G*n* suboptions you choose, the resulting binary will run on any Intel Architecture 32-bit (IA-32) processor. The following are the suboptions for -G*n* :

G4	Choose G4 when the binary application runs primarily on the Intel486 processor.
G5	Choose G5 when the binary application runs primarily on the Pentium processor.
G6	Choose G6 when the binary application runs primarily on the Pentium Pro or Pentium II processors. The G6 switch is ON by default.

For example, the following commands both compile the source program prog.cpp and optimize for the Pentium Pro and Pentium II processors:

```
prompt> icl prog.cpp
prompt> icl -G6 prog.cpp
```

Automatic Processor Dispatch Support (-Qx*[extensions]* and -Qax*[extensions]*)

The -Qx*[extensions]* and -Qax*[extensions]* options provide support to generate code that is specific to processor-instruction extensions.

-Qx*[extensions]* generates specialized code to run exclusively on the processors indicated by the extension.

-Qax*[extensions]* generates code specialized to the specified extensions, but also generates generic IA-32 code. The generic code is usually slower.

You can specify the same extensions for either option as listed in Table 4-2.

Table 4-2 Processor Dispatch Extension Options

Extension	Processors and Features Provided by the Extension
i	Pentium® Pro processor and Pentium II processors, which use the CMOV and FCMOV instructions
M	Pentium Pro processors with MMX™ technology instructions
K	Pentium III processor with the Streaming SIMD Extensions, which include the i and M instruction sets as well

Exclusive Specialized Code with -Qx*[extensions]*

The -Qx*[extensions]* option specifies the minimum extension set required to exist on processors on which you execute your program. The resulting code can contain unconditional use of the specified processor extensions. When you use -Qx*[extensions]*, the code generated by the compiler cannot execute on IA-32 processors that lack the specified extensions. The following example compiles the program contained in myprog.cpp, using the i extension:

```
prompt> icl -O2 -G6 -Qxi myprog.cpp
```

The resulting program, myprog.exe, might not execute on a Pentium processor, but will execute on Pentium Pro and Pentium II processors.

CAUTION. *A program compiled with -Qx[extensions] can fail with an Illegal instruction exception, or other unexpected behavior, if it is executed on a processor that lacks the specified extensions.*

Specialized and Generic Code with -Qax*[extensions]*

Use -Qax*[extensions]* to enable automatic code specialization for a specific set of processors extensions. The compiler generates code to detect, at runtime, the extensions supported by the current processor. Specialized code is executed if the applicable extensions are supported by the processor. The compiler also generates an equivalent, but likely slower, version of code that is generic. Thus, programs compiled with -Qax*[extensions]* execute on any IA-32 processor, subject to any restrictions imposed if you also compile with the -Qx*[extensions]*.

The following command compiles the program contained in `myprog.cpp`, and allows the compiler to generate specialized code to exploit the architectural features of the Pentium Pro processor such as the `CMOV` instruction and the MMX™ technology instructions:

```
prompt> icl -O2 -G6 -QaxiM myprog.cpp
```

The resulting program, `myprog.exe`, executes on all IA-32 processors.

CAUTION. *When you use `-Qax[extensions]` in conjunction with `-Qx[extensions]`, the extensions specified by `-Qx[extensions]` can be used unconditionally by the compiler, and the resulting program will require the processor extensions to execute properly.*

Prefetching with -Qpf*[options]*

Use `-Qpf` to automatically insert prefetching on a Pentium III processor. This option enables three suboptions described in the list that follows, each of which seeks to improve cache behavior. The following example invokes `-Qpf` as one option with all its functionality:

```
prompt> icl -Qpf[options] a.cpp
```

Invoke the following suboptions to provide only some of the functionally:

`-Qpf_loop` Automatically inserts prefetch instructions for array references that have as their subscript a simple linear function of the index variable. In addition it will improve this prefetching efficiency by unrolling loops so that 1 prefetch is executed for each iteration of the resulting loop or the size of the loop body is sufficiently large to cover the possible latency of a prefetch. The number of iterations unrolled depends on the size of the data elements, the stride of the loop and the size of the loop body.

`-Qpf_call` When you make a call to a function with a pointer to a `struct` as an argument, `-Qpf` assumes that the struct will be used within the function, and so it generates a prefetch of that `struct`.

`-Qpf_sstore` Enables the generation of streaming store instructions. Streaming store instructions reduce the chance of cache pollution, and can improve usage of the memory bus.

Loop Unrolling with -Qunroll*n*

Use `-Qunroll`*n* to specify the maximum number of times you want to unroll a loop. The following example unrolls a loop at most four times:

```
prompt> icl -Qunroll4 a.cpp
```

To disable loop unrolling, specify *n* as `0`. The following example disables loop unrolling:

```
prompt> icl -Qunroll0 a.cpp
```

Inline Expansion of Library Functions (-Oi, -Oi-)

By default, the compiler inlines a number of standard C, C++, and math library functions. This usually results in faster execution of your program.

Sometimes inline expansion of library functions can cause unexpected results. The inlined library functions do not set the `errno` variable. So in code that relies upon the setting of the `errno` variable, you should use the `-Oi-` option. Also, if one of your functions has the same name as one of the compiler's supplied library functions, the compiler assumes that it is one of the latter and replaces the call with the inlined version. Consequently, if the program defines a function with the same name as one of the known library routines, you must use the `-Oi-` option to ensure that the program's function is the one used.

For example, use `-Oi-` to ensure the disabling of inline expansion with the following options:

- IEEE 754 conformance (`-Op`)
- ANSI standard conformance (`-Za`)
- restrict optimization (`-Od`)
- debugging information (`-Zi`)

Your results can vary slightly using the preceding optimizations.

NOTE. *Automatic inline expansion of library functions is not related to the inline expansion that the compiler does during interprocedural optimizations. For example, the following command compiles the program* `sum.cpp` *without expanding the library functions:*

```
prompt> icl -Qip -Oi- sum.cpp
```

Floating-point Arithmetic Precision (-Op, -Op-, -Qprec, -Qprec_div, -Qpc, -Qlong_double)

The options described in this section all provide optimizations with varying degrees of precision in floating-point arithmetic.

When to Use -Op

The `-Op` option restricts optimization to maintain declared precision and to ensure that floating-point arithmetic conforms more closely to the ANSI and IEEE standards.

For most programs, specifying this option adversely affects performance. If you are not sure whether your application needs this option, try compiling and running your program both with and without it to evaluate the effects on performance versus precision.

Specifying this option has the following effects on program compilation:

- User variables declared as floating-point types are not assigned to registers.
- Whenever an expression is spilled, it is spilled as 80 bits (extended precision), not 64 bits (double precision).
- Floating-point arithmetic comparisons conform to IEEE 754 except for `NaN` behavior.
- The exact operations specified in the code are performed. For example, division is never changed to multiplication by the reciprocal.

- The compiler performs floating-point operations in the order specified without reassociation.
- The compiler does not perform the constant-folding optimization on floating-point values. Constant folding also eliminates any multiplication by 1, division by 1, and addition or subtraction of 0. For example, code that adds 0.0 to a number is executed exactly as written. Compile-time floating-point arithmetic is not performed to ensure that floating-point exceptions are also maintained.
- Floating-point operations conform to ANSI C. When assignments to type `float` and `double` are made, the precision is rounded from 80 bits (extended) down to 32 bits (`float`) or 64 bits (`double`). When you do not specify `-Op`, the extra bits of precision are not always rounded before the variable is reused.
- The `-Oi-` option, which disables inline functions expansion, is used.

NOTE. *The `-Oi-` and `-Op` options are active by default when you choose the `-Za` (strict ANSI C conformance) option.*

When to Use -Qprec

Use the `-Qprec` option to improve floating-point precision. `-Qprec` disables fewer optimizations and has less impact on performance than `-Op`.

When to Use -Qprec_div

Use `-Qprec_div` to disable the floating point division-to-multiplication optimization. The Intel C/C++ Compiler can change floating-point division computations into multiplication by the reciprocal of the denominator. This change can alter the results of floating point division computations slightly.

When to Use -Qpc

Use the `-Qpc` *nn* option to enable floating-point significand precision control. Some floating-point algorithms are sensitive to the accuracy of the significand or fractional part of the floating-point value. For example,

iterative operations like division and finding the square root can run faster if you lower the precision with the `-Qpc` *nn* option. Set *nn* to one of the following values to round the significand to the indicated number of bits:

`-Qpc 32`	24 bits (single precision)
`-Qpc 64`	53 bits (double precision)
`-Qpc 80`	64 bits (extended precision)

The default value for `nn` is 64, indicating double precision.

This option allows full optimization. Using this option does not have the negative performance impact of using the `-Op` option because only the fractional part of the floating-point value is affected. The range of the exponent is not affected.

The `-Qpc` *nn* option causes the compiler to change the floating point precision control when the `main()` function is compiled. The program that uses `-Qpc` *nn* must use `main()` as its entry point, and the file containing `main()` must be compiled with `-Qpc` *nn*.

Alternatives to the -Qpc Option

The `-Qpc` *nn* option performs its intended function if your program has a `main()` routine and only if you use the Intel compiler to compile the file that contains `main()`. Otherwise, this option is not useful.

An alternative method to achieve the effect of `-Qpc` *nn*, which does not require the use of the Intel compiler to compile `main()`, is through the use of the Microsoft C library routine `_controlfp()`. The `_controlfp()` function enables floating-point precision modification. To use this routine, you must include the Microsoft header file `float.h`. Table 4-3 summarizes how to achieve the effects of `-Qpc nn` using `_controlfp()`.

Table 4-3 -Qpc Equivalent Calls

Option	Equivalent Call	Extra Options
-Qpc 32	_controlfp(_PC_24, _MCW_PC);	none
-Qpc 64	_controlfp(_PC_53, _MCW_PC);	none
-Qpc 80	_controlfp(_PC_64, _MCW_PC);	none

NOTE. *For better performance, it is preferable to insert the call to* `_controlfp()` *within an initialization routine in your program.*

Figure 4-1, shows how to use `_controlfp()` in a program. See the Microsoft On-line Help for more information on `_controlfp()`.

Figure 4-1 Example of the _controlp() Function:

```
// init.c - contains initialization code for my application
#include <float.h>
void init_program(void){
// Set FPU precision control to use 24-bit significand
_controlfp( _PC_24, _MCW_PC );

// other initialization code follows...
}
```

When to Use -Qlong_double

Use the `-Qlong_double` to change the size of the `long double` type to 80-bits. For compatibility with the Microsoft compiler, the Intel compiler's default `long double` type is 64 bits in size, the same as the `double` type. This option introduces a number of incompatibilities with other files compiled without this option and with calls to library routines. Therefore, Intel recommends that the use of `long double` variables be local to a single file when you compile with this option.

Rounding Control Option (-Qrcd)

The Intel compiler uses the `-Qrcd` option to improve the performance of code that requires floating point calculations. The optimization is obtained by controlling the change of the rounding mode.

The system default floating point rounding mode is round-to-nearest. This means that values are rounded during floating point calculations. However, the C language requires floating point values to be truncated when a

conversion to an integer is involved. To do this, the compiler must change the rounding mode to truncation before each floating point conversion and change it back afterwards.

The `-Qrcd` option disables the change to truncation of the rounding mode in floating point-to-integer conversions. This means that all floating point calculations must use the default round-to-nearest, including floating point-to-integer conversions. This option has no effect on floating point calculations, but conversions to integer will not conform to C semantics.

Interprocedural and Profile Guided Optimizations

5

This chapter describes two methods to improve the performance of your code based on its unique profile and procedural dependencies.

Interprocedural Optimization (IPO)—Use the `-Qip` option to analyze your code and apply optimizations between procedures within each source file. Use multifile IPO with `-Qipo` to enable the optimizations between procedures in separate source files.

Profile Guided Optimization (PGO)—Creates an instrumented program from your source code and special code from the compiler. Each time this instrumented code is executed, the compiler generates a dynamic information file. When you compile a second time, the dynamic information files are merged into a summary file. Using the profile information in this file, the compiler attempts to optimize the execution of the most heavily travelled paths in the program.

Unlike other optimizations such as those strictly for size or speed, the results of IPO and PGO vary. This is due to each program having a different profile and different opportunities for optimizations. The guidelines provided help you determine if you can benefit by using IPO and PGO. But, only you can make this evaluation by understanding the principles of the optimizations and the unique aspects of your source code.

CAUTION. *The* `-Qip` *and* `-Qipo` *options can in some cases significantly increase compile time and code size.*

Interprocedural Optimization (IPO) with -Qip and -Qipo

Use `-Qip` to enable interprocedural optimizations, which allow the compiler to analyze your code to determine where you can benefit from the optimizations listed in Table 5-1.

Table 5-1 IPO Optimization Summary

Optimization	Affected Aspect of Program
inline function expansion	calls, jumps, branches, and loops
interprocedural constant propagation	arguments, global variables, and return values
passing arguments in registers	calls, register usage
loop-invariant code motion	further optimizations, loop invariant code
dead code elimination	code size
propagation of function characteristics	call deletion and call movement

Inline function expansion is one of the main optimizations performed by the interprocedural optimizer. For function calls that the compiler believes are frequently executed, the compiler might decide to replace the instructions of the call with code for the function itself.

With `-Qip`, the compiler performs inline function expansion for calls within procedures defined within the current source file. However, when you use `-Qipo` to specify multifile IPO, the compiler performs inline function expansion for calls between procedures defined in separate files.

Multifile IPO (-Qipo)

Multifile IPO obtains potential optimization information from individual program modules of a multifile program. Using the information, the compiler performs optimizations across modules. The result of `-Qipo` is a set of information files with a `.il` suffix.

Building a program is divided into two phases: compilation and linkage. Multifile IPO performs different work depending on whether the compilation or linkage is being performed.

1. *Compilation Phase*—As each source file is compiled, multifile IPO creates an information file. This file has the same name as the object file but with a `.il` suffix. The information file contains an intermediate representation of the source code as well as summary information used for optimization.The compiler also produces a `.obj` file when multifile IPO is enabled.
2. *Linkage Phase*—When you specify `-Qipo`, the compiler is invoked a final time before the linker. The compiler performs multifile IPO across all modules for which a corresponding up-to-date `.il` file exists.

Creating a Multifile IPO Executable

To following two steps show how to enable multifile IPO:

1. Compile your modules with `-Qipo`; as follows:
   prompt> **icl -Qipo -c a.cpp b.cpp c.cpp**

   Use `-c` to stop compilation after generating the `.il` and `.obj` files.
   Resulting information files: `a.il`, `b.il`, and `c.il`
   Resulting object files: `a.obj`, `b.obj`, and `c.obj`

NOTE. *The path and name of the `.il` file must match the object file.*

2. With preceding results, you can now optimize interprocedurally:
   prompt> **icl -Fe*nu_ip_ofile* -Qipo a.obj b.obj c.obj**

   The `-Fe` option stores the executable in *`nu_ipo_file.exe`*.
   Multifile IPO is applied only to modules with up-to-date `.il` files, otherwise the object file passes to link stage.

For efficiency, combine steps 1 and 2:

prompt> **icl -Qipo -Fe*nu_ipofile* a.cpp b.cpp c.cpp**

See "Using Profile-Guided Optimization: An Example" later in this chapter for a description of how to use multifile IPO with profile information for further optimization.

Creating a Multifile IPO Executable Using a Project Makefile

Most applications use a make file or something similar to call a linker such as `link.exe`. This is done automatically when you compile and link with `icl`. Therefore, when `-Qipo` must result in a separate linking step, you must use the Intel linker driver `xilink.exe` instead, as follows:

`prompt>` **`xilink -Qipo`** *`<LINK_commandline>`*

`-Qipo` enables multifile IPO

`<LINK_commandline>` is your linker command line

Use the `xilink.exe` syntax when you use a makefile instead of step 2 in the example "Creating a Multifile IPO Executable". The following example places the multifile IPO executable in ***`filename.exe`*** :

`prompt>` **`xilink -Qipo -out:`*`filename.exe`*` a.obj b.obj c.obj`**

NOTE. *For a `-Qipo` command line that results in linkage, the mixed order of object files and linker arguments is not preserved. Instead, the specified linker arguments follow the `-Qipo` optimized object and other objects specified in the linker's command line.*

Controlling Inline Expansion of User Functions (-Ob*n*, -Qip_no_inlining)

The compiler enables you to control the amount of inline function expansion, with `-Obn`. See Table 5-2 for details.

Table 5-2 Summary of -Ob*n* and -Qip_no_inlining Options

Option	Effect
`-Ob0`	Disables inline expansion of user-defined functions.
`-Ob1`	Enables inlining of functions declared with the `__inline` keyword, and enables inlining according to the C++ language. This option is the default if `-O1`, `-O2`, or `-Ox` are specified.

Table 5-2 Summary of -Ob*n* and -Qip_no_inlining Options (Continued)

Option	Effect
`-Ob2`	Enables inlining of any function. However, the compiler decides which functions are inlined. This option enables interprocedural optimizations and has the same effect as specifying the `-Qip` option.
`-Qip_no_inlining`	This option is only useful if `-Qip` or `-Ob2` is also specified. In this case, `-Qip_no_inlining` disables inlining that would result from the `-Qip` or `-Ob2` interprocedural optimizations, but has no effect on other interprocedural optimizations. This option has no effect on user-directed inlining (`-Ob1`).

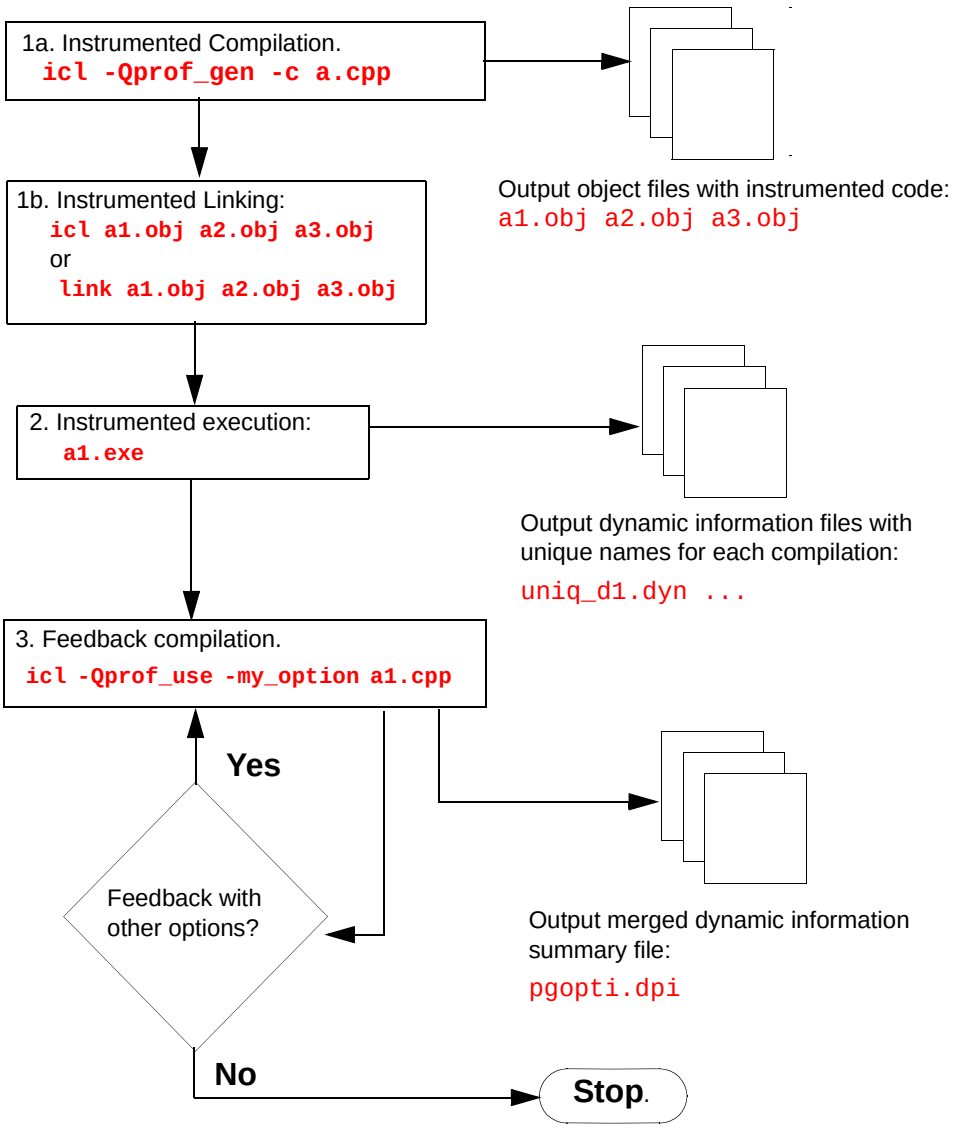
Profile-Guided Optimization (PGO): Three Phases

Profile-guided optimizations tell the compiler which areas of an application are most frequently executed. By knowing these areas, the compiler is able to be more selective and specific in optimizing the application. For example, the use of PGO often allows the compiler to make better decisions about function inlining, thereby increasing the effectiveness of interprocedural optimzations. The PGO methodology requires three phases:

1. *instrumentation compilation* and linking with `-Qprof_gen`
2. *instrumented execution* by running the executable
3. *feedback compilation* with `-Qprof_use`

These steps are summarized in Figure 5-1, followed by option descriptions and an example. A key factor in deciding whether you want to use PGO lies in knowing which sections of your code are the most heavily used. If the data set provided to your program is very consistent and it elicits a similar behavior on every execution, then PGO can probably help optimize your program execution. However, different data sets can elicit different algorithms to be called. This can cause the behavior of your program to vary from one execution to the next.

Figure 5-1 Phases of Basic Profile Guided Optimization



In cases where your code behavior differs greatly between executions, you have to ensure that the benefit of the profile information is worth the effort required to maintain up-to-date profiles. Only two options are used in a basic profile-guided optimization. These options are `-Qprof_gen` and `-Qprof_use`, used in phases 1 and 3 as described in Table 5-3.

Table 5-3 Basic Profile-Guided Optimization Options

Option	Description
`-Qprof_gen`	Instructs the compiler to produce instrumented code in your object files in preparation for instrumented execution. *Note:* The dynamic information files are produced in phase 2 when you run the executable.
`-Qprof_use`	Instructs the compiler produce a profile-optimized executable and merges available dynamic information (`.dyn`) files into a `pgopti.dpi` file. If you perform multiple executions of the instrumented program, `-Qprof_use` merges the dynamic information files again and overwrites the previous `pgopti.dpi` file.

Basic PGO Options and Environment Variables

Table 5-4 describes environment values to determine the directory in which to store dynamic information files or whether to overwrite `pgopti.dpi`. Refer to your operating system documentation for instructions on how to specify environment values.

Table 5-4 Profile-Guided Optimization Environment Variables

Variable	Description
`PROF_DIR`	Specifies the directory in which dynamic information files are created. This variable applies to all three phases of the profiling process.
`PROF_NO_CLOBBER`	Alters the feedback compilation phase slightly. By default, during the feedback compilation phase, the compiler merges the data from all dynamic information files and creates a new `pgopti.dpi` file, even if one already exists. When this variable is set, the compiler does not overwrite the existing `pgopti.dpi` file. Instead, the compiler issues a warning and you must remove the `pgopti.dpi` file if you want to use additional dynamic information files.

Using Profile-Guided Optimization: An Example

The following is an example of the basic PGO phases:

1. ***Instrumentation Compilation and Linking***—Use `-Qprof_gen` to produce an executable with instrumented information; for example:

   ```
   prompt> icl -Qprof_gen -c a1.cpp a2.cpp a3.cpp
   prompt> icl a1.obj a2.obj a3.obj
   ```

   In place of the second command, you could use the linker (`link.exe`) directly to produce the instrumented program. If you do this, make sure you link with the `libirc.lib` library.

2. ***Instrumented Execution—***Run your instrumented program with a representative set of data to create a dynamic information file.

   ```
   prompt> a1.exe
   ```

   The resulting dynamic information file has a unique name and `.dyn` suffix every time you run `a1.exe`. The instrumented file helps predict how the program runs with a particular set of data.You can run the program more than once with different input data.

3. ***Feedback Compilation***—Compile and link the source files with `-Qprof_use` to use the dynamic information to optimize your program according to its profile:

   ```
   prompt> icl -Qprof_use -Qip a1.cpp a2.cpp a3.cpp
   ```

   Besides the optimization, the compiler produces a `pgopti.dpi` file. You typically specify the default optimizations (`-O2`) for phase 1, and specify more advanced optimizations (`-Qip` or `-Qipo`) for phase 3. This example used `-O2` in phase 1 and the `-Qip` in phase 3.

NOTE. *The compiler ignores the* `-Qip` *or the* `-Qipo` *options with* `-Qprof_gen`.

Guidelines for Using PGO

When you use PGO, consider the following guidelines:

- Minimize the changes to your program after instrumented execution and before feedback compilation. During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.

NOTE. *The compiler issues a warning that the dynamic information corresponds to a modified function.*

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.

Profile Guided Optimizations Using a Function Order List

A function order list is a text file that specifies the order in which the linker should link the non-static functions of your program. This improves the performance of your program by reducing paging and improving code locality. Profile-guided optimizations support the generation of a function order list to be used by the Microsoft Visual C++ Linker. The compiler determines the order using profile information.

To enable the Intel C/C++ compiler and `proforder` tool to generate a function order list, you must use the `-Qprof_genx` and `-Qprof_dir` described in Table 5-5.

Table 5-5 Advanced Profile-Guided Optimization Options

Option	Description
`-Qprof_genx`	Generates an instrumented object file and creates a static profile information file (`.spi`). When you use `-Qprof_genx` instead of `-Qprof_gen`, you are able to use the `proforder` tool to create a function order list for the linker. However, using `-Qprof_genx` for instrumentation requires more memory at runtime, and it produces larger `.dyn` files and you cannot execute parallel make files when you use `-Qprof_genx`

Table 5-5 Advanced Profile-Guided Optimization Options (Continued)

Option	Description
`-Qprof_dir <dirname>`	Specifies the directory where `.dyn` files are to be created. The default is the directory where the program is compiled. The specified directory must already exist. You should specify the same `-Qprof_dir` option for both the instrumentation and feedback compilations. If you move the `.dyn` files, you need to specify the new path.

You will need to use the utilities `profmerge` and `proforder` described later in this chapter in "Utilities for Profile-Guided Optimization".

Function Order List Usage Guidelines

Use the following guidelines to create a function order list.

1. The order list only affects the order of non-static functions.
2. Do not use `-Qprof_genx` to compile two files from the same program simultaneously. This means that you cannot use the `-Qprof_genx` option with parallel `makefile` utilities.
3. You must compile with `-Gy` to enable function-level linking. This option is active when you specify either option `-O1` or `-O2`.

Function Order List Example

Assume you have a C program that consists of files `file1.c` and `file2.c` and that you have created a directory for the profile data files in `c:\profdata`. Do the following to generate and use a function order list.

1. Compile your program by specifying `-Qprof_genx` and `-Qprof_dir`:
   ```
   prompt> icl -FeMYPROG -Qprof_genx -Qprof_dir
   c:\profdata file1.c file2.c
   ```
2. Run the instrumented program on one or more sets of input data.
   ```
   prompt> MYPROG.exe
   ```
   The program produces a `.dyn` file each time it is executed.
3. Merge the data from one or more runs of the instrumented program using the `profmerge` tool to produce the `pgopti.dpi` file.
   ```
   prompt> profmerge -Qprof_dir c:\profdata
   ```

4. Generate the function order list using the `proforder` tool. By default, the function order list is produced in the file `proford.txt`.

```
prompt> proforder -Qprof_dir c:\profdata
-o MYPROG.txt
```

5. Compile your application with profile feedback by specifying the `-Qprof_use` and the `/ORDER` option to the linker. Again, use the `-Qprof_dir` option to specify the location of the profile files.

```
prompt> icl -FeMYPROG -Qprof_use -Qprof_dir
c:\profdata file1.c file2.c -link /ORDER:@MYPROG.txt
```

Utilities for Profile-Guided Optimization

Use the `profmerge` and `proforder` to help you merge dynamic information files and improve the performance of your programs.

The profmerge Utility

This tool merges the dynamic profile information files (`.dyn`). The compiler executes this tool automatically during the feedback compilation phase when you specify `-Qprof_use`. You need this tool if you are generating a function order list for use with the `/ORDER` option provided by the Microsoft Visual C++ linker. The command-line usage for `profmerge` is as follows:

```
prompt> profmerge [-nologo] [-prof_dir <dir_name>]
```

This merges all `.dyn` files in the current directory or the directory specified by `-prof_dir`, and produces the summary file `pgopti.dpi`.

The proforder Utility

Generates a function order list for use with the `/ORDER` option provided by the linker. The command line syntax for this tool is as follows:

```
prompt> proforder [-nologo] [-prof_dir <dir_name>]
[-o <order_file>]
```

`<dir_name>` is the directory containing the profile files (`.dpi`, `.dyn`, and `.spi`)

`<order_file>` is the optional name of the function order list file. The default name is `proford.txt`.

The `proforder` tool is used as part of the feedback compilation phase, in conjunction with the Microsoft Visual C++ linker, to improve the performance of your program.

Function Call to Dump Profile Data Explicitly

As part of the instrumented execution phase of profile-guided optimization, the instrumented program writes profile data to the dynamic information file (`.dyn file`). The file is written after the instrumented program returns normally from `main()` or calls the standard C `exit` function. For programs that do not terminate normally, the `_PGOPTI_Prof_Dump` function is provided. During the instrumentation compilation (`-Qprof_gen`) you can add a call to this function to your program. You should add the following function prototype prior to the call:

```
void _cdecl _PGOPTI_Prof_Dump(void);
```

NOTE. *You must remove the call prior to the feedback compilation (`-Qprof_use`). You can also enable and disable the call by commenting it out depending on whether you want the dynamic information file to be overwriiten during feedback compilation.*

Specifying Compilation Output

6

This chapter describes all the Intel C/C++ options that determine the output created by the compiler. By default, the compiler converts source code directly to an executable file. However, with certain options, you can control the output file that is produced by the compiler.

By having control of the output file, you can create a file at any of the compilation phases such as assembly, object, or executable. If no errors occur during processing, you can use the output files from a particular phase as input to a later compiler invocation. Table 6-1 describes the options to control the output.

Table 6-1 Compiler Input and Output Summary

Last Phase Completed	Option	Compiler Input	Compiler Output
Preprocessing	-P, -E, or -EP	source files	preprocessed files (See "Preprocessing Only (-E, -EP, and -P)" in Chapter 7 for more information about these options.)
Syntax checking	-Zs	C or C++ source files preprocessed files	diagnostic list
Compilation	-S	source files preprocessed files	assembly-language files
Assembly	-c	source files preprocessed files assembly-language files	unlinked object files

continued

Table 6-1 Compiler Input and Output Summary (Continued)

Last Phase Completed	Option	Compiler Input	Compiler Output
Linking	(default)	source files preprocessed files assembly files object files libraries	executable file map file linkable object file
compilation, linking, or assembly	-Fa, -Fo, -Fe	source, assembly, or object files	assigns a name to an output file of your choosing

Parsing for Syntax Only (-Zs)

Use the `-Zs` option to stop processing source files after they have been parsed for C/C++ language errors. This option provides a method to quickly check whether sources are syntactically and semantically correct. The compiler creates no output file. In the following example, the compiler checks a file named `prog1.cpp`.

Any diagnostics appear on the standard error output.

```
prompt> icl -Zs prog1.cpp
```

Producing an Assembly Code Listing (-S)

Use the `-S` option to generate assembly files with the `.asm` suffix. The compiler does not perform the assembly and linking phases after the source files have been preprocessed and compiled. The file contains comments indicating the source-code line and column that corresponds to each assembly language instruction.

Use the `-Fa` option to name the resulting file. By default, the compiler uses the name of each source file with the `.asm` extension.

For example, the following command creates two assembly language files, `file.asm` and `file2.asm`, that can later be assembled and linked:

```
prompt> icl -S file.cpp file2.cpp
```

You can assemble the `.asm` files to produce `file.obj` and `file2.obj`:

```
prompt> icl -c file.asm file2.asm
```

Assembly files produced by the Intel C/C++ Compiler can be assembled using MASM 6.11 or higher. Certain additional options are required when using MASM 6.11 to assemble files produced by the Intel C/C++ Compiler. Use a command line similar to the following to call MASM:

```
ml -c -coff -Cp file.asm
```

CAUTION. *Any existing file with the same name and extension is overwritten.*

The following is an example of a portion of an assembly file listing:

```
$B1$3:      ; 100    ; preds: B1$2 B1$3
      add   eax, 4                      ; 7.5
      jne   $B1$3   ; PROB 95%          ; 8.5
                    ; LOE: eax, ebx, esi, edi, esp
```

- `$B1$3` identifies the beginning of the third basic block in the first function of the file. A basic block is a set of instructions with the property that if the first instruction is executed then all of the subsequent instructions in the set are also executed.
- `; n` following the basic block label is the block execution count. This count is only printed when the `-Qprof_use` option is used. It indicates the average number of times a block was executed when the instrumented program was run. See "Basic Profile-Guided Optimization Options" in Chapter 5 for more information on `-Qprof_use`.
- `; Preds` is a list of predecessors of the current basic block. Predecessors are blocks that can transfer control to the current basic block.
- The numbers (*x.y*) following the semicolon (`;`) at the end of each instruction indicate the source line number and column corresponding to that assembly language instruction.
- `; Prob 95%` indicates the probability assigned to a jump.

- `; LOE` indicates a list of registers which are live on exit from the current basic block. These are registers that contain values to be used by succeeding basic blocks.

Suppressing Linking (-c)

Use the `-c` option to suppress linking. For example, entering the following command produces the object files `file.obj` and `file2.obj`:

```
prompt> icl -c file.cpp file2.cpp
```

NOTE. *The preceding command does not link these files to produce an executable file.*

Using the Microsoft Assembler to Produce Object Code (-Quse_asm)

Use the `-Quse_asm` option to generate assembly code from input source files and then call the Microsoft assembler (MASM) to generate the object files. By default, the compiler generates an object file directly without going through the assembler.

For example, the following command generates assembly code, then calls the assembler to generate object files:

```
prompt> icl -c -Quse_asm file1.cpp
```

NOTE. *MASM does not support all the features used on the object files for C++. Therefore you cannot assemble all files generated by* `icl` *with MASM. See the MASM documentation for details.*

MASM 6.11 handles a maximum of approximately 32760 lines. If you attempt to assemble a `.asm` file with more than 32760 lines, the assembler aborts with a fatal error. Also, if you compile with the `-Quse_asm` option and the intermediate assembly file exceeds 32760 lines, the assembler aborts.

Assembly files are produced by the -S option and are implicitly produced and assembled by the -Quse_asm option. Due to limitations in MASM and the assembly file format, it may not be possible to assemble .asm files produced by the Intel C/C++ Compiler. The following are specific features that are incompatible with assembly file usage:

- Thread Local Storage using __declspec(thread) syntax
- Debug Information Generation using the -Zi, -Z7, or -Zd options.
- #pragma comment and the embedding of default library directives in the object file. The following options are affected: -LD, -LDd, -MD, -MDd, -ML, -MLd, -MT, -MTd, -V
- COMDAT generation using the -Gf or -Gy options is not supported in assembly files.
- Building of multifile C++ programs is not possible when assembly files are used, due to lack of support for COMDAT generation.

Linking

If you do not specify any of the preceding phase limitation options, the compiler defaults to creating executable files; providing that your code is error-free. The following sections describe options that you can use to change the name of the output file and how to prepare for debugging.

Naming the Output File (-Fe, -Fo, -Fa)

When compiling and linking a set of source files, you can use the -Fe, -Fo, or -Fa options to give the resulting file a name other than that of the first source or object file on the command line. If you are processing a single file, you can use the -Fe*name*, -Fo*name*, and -Fa*name* options to specify alternate names respectively for executable, object, and assembly files. Do not enter a space between the option and the *name* argument. You can also use these options to override the default filename extensions .asm and .obj. In the following example, the -Fo option assigns the name file.obj to an output object file rather than the default (x.obj). The -c option directs the compiler to suppress linking.

```
prompt> icl -c -Fofile.obj x.cpp
```

In the following example, the command produces an executable file named `outfile.exe` as the result of compiling and linking two source files.

```
prompt> icl -Feoutfile.exe file.cpp file2.cpp
```

When compiling one or more files, the `-Fe`, `-Fa`, and `-Fo` options can be given a *dirname* (that is, a directory name) argument. The *dirname* argument must end in a forward slash "`/`" or backslash "`\`" character, and it must name an existing directory. In this case, the compiler uses the default convention in naming the executable, assembly, or object files produced, but the files will be placed in the directory specified by *dirname*.

In the following example, assume that `obj_dir` is an existing directory. The `-Fe` option causes the compiler to create the executable `myprog.exe` in the current directory. The `-Fo` option causes the compiler to create the object files `a.obj`, `b.obj`, and `c.obj` and place them in the directory `obj_dir`.

```
prompt> icl -Femyprog.exe -Foobj_dir/ a.cpp b.cpp c.cpp
```

Do not enter a space between the option and the *dirname* argument. You can specify different name arguments for each of the `-Fe`, `-Fa`, and `-Fo` options. The compiler does not remove objects that it produces, even when the compilation proceeds to the link phase.

Preparing for Debugging (-Zi, -Oy, -Oy-)

Use the `-Zi` option to direct the compiler to generate code to support symbolic debugging. For example:

```
prompt> icl -Zi prog1.cpp
```

The compiler does not support the generation of debugging information in assembly files. If you specify the `-Zi` option with `-Quse_asm`, the resulting object file will not contain debugging information. If you specify the `-Zi` option with `-S` and later assemble the resulting assembly file, the resulting object file will not contain debugging information. If you specify the `-Zi` option with `-Fa`, the resulting object file will contain debugging information, but the assembly file will not.

Also, because the Intel C/C++ Compiler does not use Microsoft's PDB format for debug information, object files with debug information are larger than those produced by the Microsoft Visual C++ Compiler.

The compiler uses `-Od` as the default when you specify `-Zi`. Specifying the `-Zi` or `-Od` option automatically disables the `-Oy` option.

The `-Oy` option, which is enabled by default or when `-O1` or `-O2` is specified, allows the compiler to use the `ebp` register as a general purpose register in optimizations. However, most debuggers expect `ebp` to be used as a stack frame pointer, and cannot produce a stack backtrace unless this is so. The `-Oy-` option instructs the compiler to generate code that maintains and uses `ebp` as a stack frame pointer, without turning off optimization, so that a debugger can still produce a stack backtrace. Using this option reduces the number of available general purpose registers by one, and can result in slightly less efficient code.

Optimizations and Support for Symbolic Debugging

As described in the preceding section, specifying `-Zi` or `-Od` automatically disables the `-Oy` option. The compiler lets you generate code to support symbolic debugging while the `-O1`, or `-O2` optimization options are specified on the command line along with `-Zi`. However, you can receive these unexpected results:

- If you specify the `-O1`, or `-O2` options with the `-Zi` option, some of the debugging information returned may be inaccurate as a side-effect of optimization.
- If you specify the `-O1`, or `-O2` options, the `-Oy` option will not be disabled. In this case, if you want to maintain the frame pointer while generating debug information, you must explicitly specify the `-Oy-` option on the command line to disable `-Oy`.

The effects of using the `-Zi` option with the optimization options are summarized in Table 6-2.

Table 6-2 Effects of Using -Zi with Optimization Options

These options	Imply these results
`-Zi`	debugging information produced, `-Od`, `-Oy` disabled
`-Zi -O2`	debugging information produced, `-O2` optimizations enabled
`-Zi -O2 -Oy-`	debugging information produced, `-O2` optimizations enabled, `-Oy` disabled
`-Zi -Qip`	limited debugging information produced, `-Qip` option enabled

See the respective sections in Chapter 4, Optimizations for detailed descriptions of the optimizations options listed in Table 6-2.

Preprocessing

7

This chapter describes the options you can use from the command line to direct the operations of the preprocessor. Preprocessing performs such tasks as macro substitution, conditional compilation, and file inclusion. Table 7-1 provides a summary of the preprocessing options.

Table 7-1 Options to Control Preprocessing

Option	Description
-QA*name*[(*tokens*)]	Associates a symbol name with the specified sequence of values; equivalent to an `#assert` preprocessing directive.
-QA-, -u	Causes all predefined macros (other than those beginning with _ _) and assertions to be inactive.
-C	Preserves comments in preprocessed source output.
-D*name*[=*value*]	Defines the macro *name* and associates it with the specified *value*. The default (-D*name*) defines a macro with a *value* of 1.
-E	Directs the preprocessor to expand your source module and write the result to standard output.
-EP	Same as -E but does not include `#line` directives in the output.
-P	Directs the preprocessor to expand your source module and store the result in a file in the current directory.
-U*name*	Suppresses any automatic definition for the specified macro.

Preserving Comments in Preprocessed Source Output (-C)

Use the `-C` option to preserve comments in your preprocessed source output.

Preprocessing Only (-E, -EP, and -P)

Use either the `-E` or the `-P` option to preprocess your source files without compiling them.

When you specify the `-E` option, the compiler's preprocessor expands your source module and writes the result to standard output. The preprocessed source contains `#line` directives, which the compiler uses to determine the source file and line number during its next pass. For example, to preprocess two source files and write them to `stdout`, enter the following command:

```
prompt> icl -E prog1.cpp prog2.cpp
```

When you specify the `-P` option, the preprocessor expands your source module and stores the result in a file in the current directory. There is no way to change the default name. The preprocessor uses the name of each source file with the `.i` extension. For example, the following command creates two files named `prog1.i` and `prog2.i`, which you can use as input to another compilation:

```
prompt> icl -P prog1.cpp prog2.cpp
```

The `-EP` option can be used in combination with `-E` or `-P`. It directs the preprocessor to not include `#line` directives in the output. Specifying `-EP` alone is the same as specifying `-E -EP`.

CAUTION. *When you use the `-P` option, any existing files with the same name and extension are overwritten.*

Defining Macros (-QA, -QA-, -u, -D, and -U)

You can use the -QA and -D options to define the assertion and macro names to be used during preprocessing. The -U option directs the preprocessor to suppress an automatic definition of a macro.

Use the -QA option to make an assertion. This option performs the same function as the #assert preprocessor directive. The form of this option is:

-QAname[(value)]

name	Indicates an identifier for the assertion.
value	Indicates a value for the assertion. If a value is specified, it should be quoted, along with the parentheses delimiting it.

For example, to make an assertion for the identifier fruit with the value orange,banana, use the following command:

```
prompt> icl -QA"fruit(orange,banana)" prog1.cpp
```

The compiler provides a number of predefined macros. For a list of predefined macros available to the Intel C/C++ Compiler , see "Predefined Macros" .

Enter -QA- or -u to suppress all predefined macros, except for those beginning with the double underscore.

Use the -D option to define a macro. This option performs the same function as the #define preprocessor directive. The form of this option is:

-Dname[=value]

name	The name of the macro to define.
value	Indicates a value to be substituted for *name*. If you do not enter a value, *name* is set to 1. The value should be quoted if it contains non-alphanumerics.

For example, to define a macro called SIZE with the value 100 use the following command:

```
prompt> icl -DSIZE=100 prog1.cpp
```

Use the -U*name* option to suppress any automatic definition for the specified name. The -U option performs the same function as a #undef preprocessor directive.

For more details about preprocessor directives, see a language reference such as *C: A Reference Manual*.

Predefined Macros

The predefined macros available for the Intel C/C++ Compiler are described in Table 7-2. The **Default** column describes whether the macro is enabled (ON) or disabled (OFF) by default. The **Disable** column lists the option that disables the macro; no indicates that the macro cannot be disabled.

Table 7-2 Predefined Macros

Macro Name	Default	Disable	Description / When Used
__ICL=n	n=400	no	Identifies the Intel C/C++ Compiler for Win32 systems Version 3.0 (n=300).
__cplusplus	C++ only	no	Defined when compiling C++ source.
_WIN32	ON	`-u`	Defined for Win32 applications
_MSC_EXTENSIONS	ON	`-u`	Specifies Microsoft Visual C++ language extensions.
_MSC_VER=n	n=1100	`-u`	Defined for Microsoft Visual C++ compatibility. Set by the installation program through options in the `icl.cfg` file. The value of `n` is based on the version of Visual C++ installed: ***version n=*** 4.0 1000 4.1 1010 4.2 1020 5.0 1100 6.0 1200

continued

Table 7-2 Predefined Macros (Continued)

Macro Name	Default	Disable	Description / When Used
_M_IX86=*n*	ON, *n*=600	`-u`	defined based on the processor option you specify: *n*=300 if you specify the `-GB` or `-G3` option *n*=400 if you specify the `-G4` option *n*=500 if you specify the `-G5` option *n*=600 if you specify the `-G6` option
_DLL	OFF	`-u`	defined if you specify the `-MD` option
_MT	OFF	`-u`	defined if you specify the `-MD`, `-MT`, or `-LD` option
_CHAR_UNSIGNED	OFF	`-u`	defined if you specify the `-J` option
_CPPRTTI	OFF	`-u`	defined if you specify the `-GR` option for C++ only
_CPPUNWIND	OFF	`-u`	defined if you specify the `-GX` option for C++ only

See "Predefined Macros for Standard Conformance" in Chapter 8 for the list of predefined macros required for ANSI conformance standard.

Printing Include-file Dependencies (-QH)

Use the `-QH` option to print the pathname for each file compiled into the source with a `#include` directive. Pathnames can be absolute or relative. The compiler displays the dependencies on the standard output and prints the path for each included file on a separate line. The compilation process stops after preprocessing is completed.

In the following example, the source file, `dtest.cpp`, includes three other files. Enter the following command to display the dependent files:

```
prompt> icl -QH dtest.cpp
./d1.h
./d2.h
./d3.h
```

Printing Makefile Dependencies (-QM)

Use the `-QM` option to generate list of makefile dependency lines for each source file in the compilation. The compiler displays the dependency lines based on `#include` lines that appear in each source file. For example, the output of a simple program with one include file might be as follows:

```
hello.obj: hello.c
hello.obj: c:/Msdev/Include/stdio.h
```

C/C++ Language Features

8

This chapter describes the C and C++ language implementation of the Intel C/C++ Compiler. This chapter does not teach you how to program in C or C++. Instead, it covers the following topics:

- conformance to the ANSI/ISO standard for C
- options that support the C++ language

Conformance to C Standards

The compiler accepts two C language dialects: strict ANSI conformance, and extended. You can select the style that best suits your application.

The compiler's implementation of C conforms to the ANSI/ISO standard (ISO/IEC 9899:1990) for the C language. The standard specifies that a conforming implementation of a C compiler must meet minimum requirements for certain translation limits. In all cases, the compiler exceeds these limits. Appendix A, "Compiler Limits," lists the tested values. The compiler also accepts extensions to the standard.

By default, the compiler does not flag all extensions with error or warning messages. You must use the `-Za` option for the compiler to strictly enforce the standard. For more information on this option, see "Strict ANSI Dialect (-Za)" in this chapter.

C Language Dialects

The compiler supports two C language dialects: strict ANSI, and extended. By default, the compiler accepts the extended dialect. You can use the `-Za` option to select the language as defined by the ANSI/ISO standard (ISO/IEC 9899:1990) with no deviation.

The following sections describe each dialect. See *C: A Reference Manual*, or *The C Programming Language*, all listed in the "About This Manual" section, for a full description of the C language.

Strict ANSI Dialect (-Za)

Use the `-Za` option to select the strict ANSI dialect. The compiler does not deviate from the standard when this option is in effect. While following strict conformance to the standard, the compiler flags any nonstandard code in the source file with warnings or error messages.

Extended Dialect (-Ze)

The compiler accepts the language specified in the standard when you enter the `-Ze` option on the invocation line, with extensions in the areas described below.

Extensions for files and data storage:

- The input file can contain no text, so you can use an empty file as input to the compiler.
- The last member of a structure can have an incomplete array type. However, that member cannot be the only member of the structure, otherwise, the structure size would be zero.
- A file-scope array can have an incomplete `struct` or `union` type as its element type. The `struct` or `union` type must be completed before the array is subscripted and by the end of the compilation if the array is defined in the compilation.
- You can declare an `enum` tag name, and then define it later in the source file.
- An initializer expression that is a single value, and is used to initialize an entire static array, structure, or union, need not be enclosed in braces. Standard C requires the braces.

- The compiler generates a remark message for a storage-class specifier appearing anywhere other than the first position in a list of specifiers, as in `int static`.

Extensions for pointers:

- In an initializer, you can cast a pointer constant value to an integral type if the integral type is big enough to contain it.
- In an integral constant expression, you can cast an integer constant to a pointer type and then back to an integral type.
- The compiler accepts assignments of pointers to integers and to other incompatible pointer types without an explicit cast.
- You can select a field in the form `p->field`, even if `p` does not point to a `struct` or `union` that contains `field`. The `p` variable must be a pointer. Likewise, `x.field` is allowed even if `x` is not a structure or union that contains `field`. The `x` variable must be an `lvalue`. For both cases, all definitions of `field` as a field must have the same type and offset within their structure or union.

Extensions for types and syntax:

- Bit fields can have `enum` base types, or integral types other than `int` and `unsigned int`.
- You can use `long float` as a synonym for `double`.
- You can place arbitrary text at the end of preprocessing directives.
- Numbers are scanned according to the syntax for numbers rather than the `pp-number` syntax. Thus, `0x123e+1` is scanned as three tokens instead of one invalid token. Under strict ANSI conformance, the compiler uses the `pp-number` syntax. See the standard text for more information on the `pp-number` syntax.

Extensions for predicates:

- You can use the `#assert` and `#unassert` macros to define and test predicate names.

Extensions with Warnings. When you use the `-Ze` option, the compiler accepts and tags the extensions in syntax and semantics as described below; however, these extensions are tagged with a warning message.

Extensions for syntax with warnings:

- You can have an extra comma at the end of an `enum` list.
- You can omit the final semicolon preceding the closing brace (`}`) of a structure or union.
- A right brace can immediately follow a label definition. Normally, a statement must follow a label definition.
- You can have an empty declaration (a semicolon with nothing before it).

Extensions for semantics with warnings:

- You are allowed assignment and pointer differences between pointers to types that are interchangeable but not identical, such as `unsigned char *` and `char *`. However, the compiler issues a warning. Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- You can compare a pointer to a `void` to a pointer of another kind using the `>`, `>=`, `<`, or `<=` operators without using an explicit type cast. Standard C allows such comparisons using `==` or `!=` and issues no warnings.
- You can insert inline assembly code using the `asm` keyword. In the strict ANSI C dialect, such insertions are disabled.
- You can have freestanding tag declarations in the parameter declaration list for a function with old-style parameters.
- The compiler generates a warning if an overflow is detected while folding signed integer operations on constants.

Predefined Macros for Standard Conformance

The standard requires that certain predefined macros be supplied with the compiler. Table 8-1 lists the macros the compiler defines in accordance with the standard.

Table 8-1 Predefined Macros for Standard Conformance

Macro	Description
_ _DATE_ _	The date of compilation as a string literal in the form “Mmm dd yyyy”.
_ _FILE_ _	A string literal representing the name of the file being compiled.
_ _LINE_ _	The current line number as a decimal constant.
_ _STDC_ _	The constant 1 under the ANSI conformance dialect (-Za); not defined in the extended dialect.
_ _TIME_ _	The time of compilation, as a string literal in the form “hh:mm:ss”.

The compiler provides some predefined macros in addition to the predefined macros required by the standard. For a list of these macros see “Predefined Macros” in Chapter 7 of this manual and the section on the preprocessor category of the Microsoft *Visual C++ User’s Guide*.

Conformance to C++ Standards

The Intel C/C++ Compiler conforms to the Microsoft implementation of the C++ language. For more information on the Microsoft implementation of the C++ language, see the Microsoft Visual C++ documentation.

Microsoft Compatibility 9

The Intel C/C++ Compiler supports the Microsoft Visual C++ extensions to the C and C++ languages. This chapter describes the differences in the way the Intel C/C++ Compiler interprets some of the Microsoft Visual C++ extensions. The differences are in the following areas:

- Compiler Pragmas
- Microsoft compatibility option (`-Qms`)
- Microsoft Version Compatibility Options (`-Qvcn`)
- Unsupported Compiler Options
- Differences in PCH Support
- Compilation and Execution Differences
- Evaluation of Left Shift Operations
- Use of Friend Injection
- Function Declaration in Scope of Function Defined in a Namespace
- Enum Bit-Field Signedness

Compiler Pragmas

The Intel C/C++ Compiler supports the Microsoft Visual C++ pragmas with the following limitations:

- The pragma `pragma optimize` accepts a list that allows the enabling and disabling of specific optimizations. The Intel compiler ignores this list. The pragma either enables or disables all optimizations specified by options on the command line.

- The following pragmas are accepted without error but have no effect:

```
pragma auto_inline      pragma component
pragma function         pragma include_alias
pragma inline_depth     pragma inline_recursion
pragma intrinsic        pragma setlocale
pragma warning
```

For more information on pragmas, see the on-line help or the documentation that accompanies your copy of Microsoft Visual C++.

Microsoft Compatibility Option (-Qms)

In a limited number of cases, the Microsoft compiler compiles source code without generating an error while the Intel compiler generates an error for the same source code. If this occurs in a file that is part of a third party distribution (such as include files or C++ class library files), it might not be convenient to correct the error in the source file. In such cases, using the `-Qms` option with the Intel compiler might allow a successful compilation.

Microsoft Version Compatibility Options (-Q*vcn*)

You can use the Intel C/C++ Compiler with Microsoft Visual C++ Versions 4.2 or higher. Some language features are available only in Visual C++ Version 5.0 or later and other features are available only in Version 6.0. The `-Qvc`*n* option tells the compiler what version of Visual C++ you are using. Table 9-1 lists the valid `-Qvc`*n* options for each version:

Table 9-1 Microsoft Version Compatibility Options

Option	Compatibility with
`-Qvc4`	Visual C++ Version 4.x
`-Qvc5`	Visual C++ Version 5.0
`-Qvc6`	Visual C++ Version 6.0

NOTE. *The Intel C/C++ Compiler installation program automatically adds the appropriate* `-Qvcn` *option for your version of Visual C/C++ (if needed) to the Intel C/C++ Compiler configuration file (*`icl.cfg`*). Therefore, you will typically not need to use the* `-Qvcn` *option.*

Unsupported Compiler Options

The Intel C/C++ Compiler supports most of the same options as the Microsoft Visual C++ Compiler, as well as Intel-specific options. However, a small subset of Microsoft Visual C++ Compiler options are not supported by the Intel C/C++ Compiler. Most of the unsupported options, while useful for development purposes, are not required to build a working application. The unsupported options are listed in Table 9-2.

Table 9-2 List of Unsupported Microsoft Visual C++ Compiler Options

Option	Description
`-Fd`	name the PDB file used for debug information for specified source files
`-GD`	optimize for Windows DLL
`-Gi`	enable incremental compilation
`-GI`	edit and continue debugging (same effect as the `-Zi` option)
`-Gm`	enable minimal rebuild
`-Oa`	assume no aliasing
`-Ow`	assume no cross-function aliasing
`-WX`	treat warnings as errors
`-Yd`	put debug information in every object (Microsoft PCH-specific option)
`-Zg`	generate function prototypes
`-Zm`	set compiler's memory allocation limit

The Intel C/C++ Compiler issues a remark stating lack of support for many of these options, but it silently ignores the following options: `-Fd`, `-Gi`, `-Gm`, `-Yd`, and `-Zm`.

Differences in PCH Support

There are some differences in how precompiled header (PCH) files are supported between the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler. These differences include the following:

- The PCH information generated by the Intel C/C++ Compiler is not compatible with the PCH information generated by the Microsoft Visual C++ Compiler.
- The Intel C/C++ Compiler does not support PCH generation and use in the same translation unit.
- The Intel C/C++ Compiler does not generate PCH information beyond a point where a declaration is seen in the primary translation unit. The Microsoft Visual C++ compiler generates PCH information beyond this point in some situations.

Compilation and Execution Differences

While the Intel C/C++ Compiler is compatible with the Microsoft Visual C++ Compiler, some differences can prevent successful compilation. Also there can be some incompatible generated-code behavior of some source files with the Intel C/C++ Compiler. In most cases, a modification of the user source file enables successful compilation with both the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler. The differences between the compilers are listed as follows:

Inline Assembly Target Labels

When compiled by the Intel C/C++ Compiler, inline assembly target labels of `goto` statements are case sensitive. The Microsoft Visual C++ Compiler treats these labels in a case insensitive manner. For example, the Intel C/C++ Compiler issues an error when compiling the following code:

```
int func(int x)
{
    goto LAB2;
    /*   error: label  "LAB2" was  referenced
     *   but not defined
     */
    __asm lab2: mov x, 1
    return x;
}
```

However, the Microsoft Visual C++ Compiler accepts the preceding code. As a work-around for the Intel C/C++ Compiler, when a `goto` statement refers to a label defined in inline assembly, you must match the label reference with the label definition in both name and case.

Preprocessor Macro Expansion

The Intel C/C++ Compiler differs from the Microsoft Visual C++ Compiler in the way it expands preprocessor macros that are used within `#include` directives. In some case the code passes a macro as a parameter to another macro that uses the token-concatenation operator. In such a case, the macro that is used as a parameter is not expanded before concatenation. This is demonstrated in the following example:

```
#define D   var
#define Decl(d)   int my ## d ## ;
Decl(D)
```

Both compilers would preprocess the preceding source and produce the following code:

```
int myD;
```

When a similar macro is used in a `#include` directive, the Intel C/C++ Compiler behaves similarly to the preceding example. However, the Microsoft Visual C++ Compiler performs an extra preprocessing scan through the `#include` directive to produce different results. In the following example, the `D` and `F` macros, when used in the `#include` directive, are not expanded by the Intel C/C++ Compiler .

```
#define  D    sys
#define  F    stat.h

#define INC(d,f)   < ## d ## / ## f ## >

#include INC(D,F)
// Visual C++ Compiler interpretation:
//# include <sys/stat.h>
// Intel C/C++ Compiler interpretation:
// #include <D/F>
```

The Intel C/C++ Compiler issues an error for this code when it fails to find the file called `D/F`. The Microsoft Visual C++ Compiler expands the `D` and `F` macros and accepts this code. The following alternative code achieves the same result but it works with both the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler:

```
#define  D    sys
#define  F    stat.h

#define CAT5(a,b,c,d,e)   a ## b ## c ## d ## e
#define INC(d,f)          CAT5(<,d,/,f,>)

#include INC(D,F)
```

Evaluation of Left Shift Operations

The Intel C/C++ Compiler differs from the Microsoft Visual C++ Compiler in the evaluation of left shift operations where the right operand, or shift count, is equal to or greater than the size of the left operand expressed in bits. The ANSI C Standard states that the behavior of such left-shift operations is undefined, meaning a program should not expect a certain behavior from these operations. This difference is only evident when both operands of the shift operation are constants. The following example illustrates this difference between the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler:

```
int x;
int y = 1;

void func()
{
 x = 1 << 32;
    // Visual C++ Compiler generates code to set x=0
    // Intel C/C++ Compiler generates code to set x=1
 y = y << 32;
    // Visual C++ Compiler generates code to set x=1
    // Intel C/C++ Compiler generates code to set x=1
}
```

Use of Friend Injection: Not Recommended

Sometimes a class is a member of a `namespace`, and contains a friend declaration of a function not already declared. In such cases, the Intel C/C++ Compiler injects the function declaration into the `namespace` containing the class, even if the class definition is not lexically within the `namespace` definition. Microsoft Visual C++ Version 4.2 apparently "injects" the function declaration into the scope lexically containing the class definition. As a result of this difference, friendship may be bestowed on a different function, and access errors may result.

Since friend "injection" has recently been eliminated as a language feature by the C++ standardization committee, the interpretation of code which relies on the specific behavior of Microsoft Visual C++ Version 4.2. In contrast, Microsoft Visual C++ Version 5.0 does not accept source like that described at the beginning of this paragraph and reports an error diagnostic message semantically equivalent to the Intel C/C++ Compiler. To avoid such behavior, you should avoid the use of friend injection.

Declaration in Scope of Function Defined in a Namespace

In accordance with the C++ language specification, if a function declaration is encountered within a function definition, the function referenced is taken to be another member of the `namespace` of the containing function; regardless of whether the containing function definition is lexically within a `namespace` definition. The Microsoft Visual C++ Compiler takes the referenced function to be a global function (not in any `namespace`). Functions declared in global or `namespace` scopes are interpreted the same way by both the Intel C/C++ Compiler and the Microsoft Visual C++ Compiler.

Enum Bit-Field Signedness

The Intel C/C++ Compiler and the Microsoft Visual C++ Compiler differ in how they attribute signedness to bit fields declared with an `enum` type. Apparently, Microsoft Visual C++ always considers `enum` bit fields to be signed, even if not all values of the `enum` type can be represented by the bit field. On the other hand, the Intel C/C++ Compiler considers an `enum` bit field to be unsigned, unless the `enum` type has at least one `enum` constant with a negative value. In any case, the Intel C/C++ Compiler produces a warning if the bit field is declared with too few bits to represent all the values of the `enum` type.

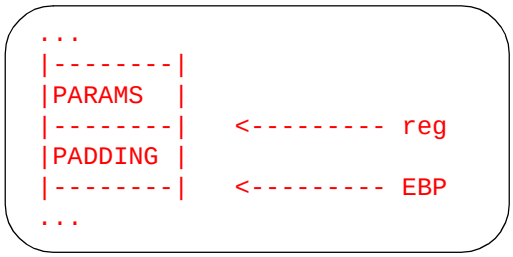
Debugging Functions with Aligned Stack Frames Using MSVC++ 4.2

When debugging a function with an aligned frame compiled with the Intel C/C++ Compiler with the MSVC++ 4.2 tools, the Visual C++ debugger displays invalid values for the parameters. This is due to the fact that the MSVC++ 4.2 linker does not support the required debug records to represent parameters in aligned frames.

Ordinarily, when entering a function, the `EBP` register points to the bottom of the parameter block. The debugger accesses function parameters using the `EBP` register plus an offset. However, in an aligned frame the `EBP`

register is decremented an arbitrary number of bytes, in order to align the frame, and a different register is used as a pointer to the parameters. See Figure 9-1.

Figure 9-1 Simplified View of Arbitrary Offset in the Stack



The Microsoft debug format supported by Visual C++ Version 4.2 does not allow flexibility in the base register used for parameter variable access, so the Visual C++ Version 4.2 debugger always uses `EBP` to access the parameters. This causes incorrect values to be displayed for the parameters, since they are not located at the expected address.

To view parameters in aligned functions, you can use assembly level debugging to look at the actual location of the parameter. The Intel C/C++ Compiler Version 4.0 uses the `EBX` register for parameter access in a function with an aligned frame. In such a function, parameters are located at the address specified by `EBX` plus an offset. Because of the padding that occurs within the parameter block to align `__m128` parameters, it might be easier to determine the correct offset by examining a reference to the parameter of interest in the disassembled code.

This problem does not affect use of the Intel C/C++ Compiler with the Visual C++ Version 5.0 tools. To use the Intel C/C++ Compiler with Visual C++ Version 5.0, you must install Visual C++ 5.0 and then reinstall the Intel C/C++ Compiler. For more information on the aligned stack frame feature, see the description of the `-Qsfalign` in "Stack Frame Alignment Options" in Chapter 13.

Diagnostic Information 10

This chapter describes the various messages that the compiler produces. These messages include the sign-on message and diagnostic messages for remarks, warnings, or errors. The compiler always displays any diagnostic message, along with the erroneous source line, on the standard output.

This chapter also describes how to control the severity of diagnostic messages.

Disabling the Sign-on Message (-nologo)

The compiler displays the following message each time that you invoke it:

```
Intel C/C++ Compiler Version x.y.z ID
Copyright (C) years Intel Corporation.  All rights reserved.
```

ID The unique identification number for this compiler.

x.y.z Identifies the version of the compiler.

years The years for which the software is copyrighted.

If you want to suppress the message, specify the `-nologo` option.

Printing the List of icl Options (-?, -help)

You can print a list of the most useful `icl` options by specifying the `-help` (or `-?`) option to the compiler. To print this list, use this command:

```
prompt> icl -help
```

or

```
prompt> icl -?
```

Diagnostic Messages

Diagnostic messages provide syntactic and semantic information about your source text. Syntactic information can include syntax errors and use of nonstandard C or C++. Semantic information includes such things as unreachable code.

Diagnostic messages can be any of the following: command-line diagnostics, remark messages, warning messages, error messages, or catastrophic error messages.

Command-line diagnostics. These messages report improper or unrecognized command-line options or arguments. Command-line error messages appear on the standard output in the form:

`icl: Command line error: message`

message	Describes the error.

Command-line warning messages appear as follows:

`icl: Command line warning: message`

Language diagnostics. These messages describe diagnostics that are reported during the processing of the source file. These diagnostics have the following format:

filename`(`*linenum*`): `*type* `[#]`*nn*`: `*message*

filename	Indicates the name of the source file currently being processed.
linenum	Indicates the source line where the compiler detects the condition.
type	Indicates the severity of the diagnostic message: `warning`, `remark`, `error`, or `catastrophic error`.
[#]nn	The number assigned to the `error` (or `warning`) message. Hard errors or catastrophes are not assigned a number.
message	Describes the diagnostic.

The following is an example of a warning message:

```
tantst.cpp(3): warning #328: Local variable “increment”
never used.
```

The compiler can also display internal error messages on the standard error. If your compilation produces any internal errors, contact your Intel representative. Internal error messages are in the following form:

```
FATAL COMPILER ERROR: message
```

Remark messages. These messages report common but sometimes unconventional use of C or C++. The compiler does not print or display remarks unless you specify level 4 for the `-W` option, as described in “Suppressing Warning Messages or Enabling Remarks (-w, -Wn)” later in this chapter. Remarks do not stop translation or linking. Remarks do not interfere with any output files. The following are some representative remark messages:

```
function declared implicitly
type qualifiers are meaningless in this declaration
controlling expression is constant
```

Warning messages. These messages report legal but questionable use of the language in the program being compiled. The compiler displays warnings by default. You can suppress warning messages by setting the diagnostic level to `0`. Warnings do not stop translation or linking. Warnings do not interfere with any output files. The following are some representative warning messages:

```
declaration does not declare anything
pointless comparison of unsigned integer with zero
possible use of = where == was intended
```

Error messages. These messages report syntactic or semantic misuse of C or C++. The compiler always displays error messages. Errors suppress object code for the module containing the error and prevent linking, but they allow parsing to continue to scan for any other errors. Some representative error messages are:

```
missing closing quote
expression must have arithmetic type
expected a “;”
```

Catastrophic errors. These messages indicate environmental problems. Catastrophic error conditions stop translation, assembly, and linking. If a catastrophic error ends compilation, the compiler displays a termination message on standard output. The following are some representative catastrophic error messages:

```
out of memory
could not open temporary file filename
could not open source file filename
```

Suppressing Warning Messages with lint Comments

The UNIX `lint` program attempts to detect features of a C or C++ program that are likely to be bugs, non-portable, or wasteful. The compiler recognizes three `lint`-specific comments: `/*ARGSUSED*/`, `/*NOTREACHED*/`, and `/*VARARGS*/`. Like the `lint` program, the compiler suppresses warnings about certain conditions when you place these comments at specific points in the source.

Suppressing Warning Messages or Enabling Remarks (-w, -W*n*)

Use the `-w` or `-Wn` option to suppress warning messages or to enable remarks during the preprocessing and compilation phases. You can enter the option with one of the following arguments:

`-W0, -w`	Displays error messages only.
`-W1, -W2, -W3`	Displays warnings and error messages. The compiler uses this level as the default.
`-W4`	Displays remarks, warnings, and error messages.

For some compilations, you might not want warnings for known and benign characteristics, such as the K&R C constructs in your code. For example, the following command compiles `newprog.cpp` and displays compiler errors, but not warnings:

```
prompt> icl -W0 newprog.cpp
```

Controlling the Severity of Diagnostics (-Qwd, -Qwr, -Qww, -Qwe)

You can control the severity of some of the diagnostics returned by the compiler. The compiler returns two types of diagnostics:

hard errors	Diagnostics issued for code that is definitely wrong or questionable. The severity of a hard error is not configurable. For hard errors, the message number is never printed. Remarks and warnings are never considered hard errors.
soft diagnostics	All other diagnostics (including remarks and warnings). For soft diagnostics, the message number is always printed. The severity of a soft diagnostic is configurable by the options described below.

In the descriptions below, *tag* represents the number associated with the diagnostic. Multiple tags are permitted, separated by commas.

`-Qwd[tag,...]`	Disable the soft diagnostics that corresponds to *tag*.
`-Qwr[tag,...]`	Override the severity of the soft diagnostics corresponding to *tag* and make it a remark.
`-Qww[tag,...]`	Override the severity of the soft diagnostics corresponding to *tag* and make it a warning.
`-Qwe[tag,...]`	Override the severity of the soft diagnostics corresponding to *tag* and make it an error.

For example, the following command line disables soft diagnostic `68` during compilation of the file `a.cpp`:

```
prompt> icl -Qwd68 -c a.cpp
```

The following command line changes the severity of soft diagnostics `68` and `152` to remarks during compilation of the file `a.cpp`.

```
prompt> icl -Qwr68,152 -c a.cpp
```

Example. Assume that you have a file `x.cpp` that contains the following lines:

```
/*
/* This is a comment
*/

extern i;
```

If you compile this code with warnings enabled (the default), you will receive the following response from the compiler:

```
x.cpp(2): warning #9: nested comment is not allowed
   /* This is a comment
   ^
x.cpp(5): warning #260: explicit type is missing ("int"
assumed)
   extern i;
   ^
```

If you compile the code with the option `-Qwd9`, (to disable warning number 9), you will receive the following response from the compiler:

```
x.cpp(5): warning #260: explicit type is missing ("int"
assumed)
   extern i;
   ^
```

Limiting the Number of Errors Reported (-Qw*num*)

Use the `-Qwnum` option to limit the number of error messages displayed before the compiler aborts. By default, if more than 100 errors are displayed, compilation aborts.

`-Qw`*num* — Limit the number of error diagnostics that will be displayed prior to aborting compilation to *num*. Remarks and warnings do not count towards this limit.

For example, the following command line specifies that if more than 50 error messages are displayed during the compilation of `a.cpp`, compilation aborts.

```
prompt> icl -Qwn50 -c a.cpp
```

Additional Information about the Compilation

The compilation system issues a variety of self-explanatory information messages about the compilation system components and the process that are not described in this manual.

Libraries 11

The compiler allows you to use all the standard run-time libraries that are part of Microsoft Visual C++ Version 4.0 or later. You can determine the libraries used by your applications by controlling the directories which the Microsoft linker searches by using the options described in this chapter.

Managing Libraries

The `LIB` environment variable contains a semicolon-separated list of directories in which the Microsoft linker will search for library (`.lib`) files. If you want the linker to search additional libraries, you can add their names to the command line, to a response file, or to the `icl.cfg` file. In each case, the names of these libraries are passed to the linker before the names of the Intel libraries (`libm.lib` or `libm_chk.lib`) and the Microsoft-provided default libraries that the driver always specifies (`oldnames.lib` and `libc.lib`). For more information on adding library names to the response file and the configuration file (`icl.cfg`), see "Response Files" and "Configuration Files" in Chapter 3.

To specify a library name on the command line, you must first add the library's path to the `LIB` environment variable. Then, to compile `file.cpp` and link it with the library `mylib.lib` enter the following command:

```
prompt> icl file.cpp mylib.lib
```

The compiler driver, `icl.exe`, passes file names to the Microsoft linker in the following order:

1. the object file
2. any objects or libraries specified on the command line, in a response file, or in a configuration file
3. the `libm.lib` library by default (or, if you specified the `-QIfdiv` option, the `libm_chk.lib` libraries)
4. the Microsoft-provided libraries `libc.lib` and `oldnames.lib`

Default Libraries

The compiler allows you to use all the standard run-time libraries that are part of Microsoft Visual C++ Version 4.x or higher. By default, the compiler automatically expands a number of standard C, C++, and math library functions. For more information, see"Inline Expansion of Library Functions (-Oi, -Oi-)" in Chapter 4.

Library Files

The compiler automatically tells the linker to use the following support libraries.

`libm_chk.lib` or `libm.lib`	The `libm.lib` file contains the math library. The `libm_chk.lib` file contains the math library with support routines for a floating-point division software patch for certain steppings of the Pentium processor. For information on these libraries, see "Enabling the Floating-Point Division Check (-QIfdiv)".

These libraries are supplied with Microsoft Visual C++ Version 4.0 or later.

`libc.lib`	The standard C run-time library.
`oldnames.lib`	Contains aliases for non-ANSI functions that normally begin with an underscore in Microsoft libraries.

If you want to link your program with alternate or additional libraries, specify them at the end of `icl` command line. For example, to compile and link `hello.cpp` with `mylib.lib`, use the following command:

```
prompt> icl -Fehello.exe hello.cpp mylib.lib
```

The `mylib.lib` library appears prior to the `libc.lib` library in the command line for the `LINK` linker.

Math Libraries

In the compiler package, you received the Intel math library, `libm.lib`, which contains optimized versions of the math functions in the standard C run-time library. The functions in the library are optimized for program execution speed on the Pentium processor.

To use the optimized math library, the set up process places `libm.lib` in one of the directories specified in the library search path defined by the `LIB` variable. Intel recommends that you keep `libm.lib` in the first directory specified in the path.

Enabling the Floating-Point Division Check (-QIfdiv)

The `-QIfdiv` option enables a software patch for the floating-point division flaw that exists on some steppings of the Pentium processor. This patch ensures that the precision of your floating-point division calculations are correct.

NOTE. *This option (`-QIfdiv`) is no longer enabled by default when you specify either the `-G3` or the `-G5` options.*

When the `-QIfdiv` option is enabled, the compiler uses `libm_chk.lib` instead of `libm.lib`. to link your programs. The `libm_chk.lib` library contains the support routines for the floating-point division software patch for affected math library functions.

The `-QIfdiv-` option disables the software patch for the floating-point division flaw regardless of whatever other options are specified. When you specify `-QIfdiv-`, the compiler uses simple hardware instructions for floating-point division and affected intrinsics. If you specify the `-QIfdiv-` option, the compiler links with `libm.lib`. Similarly, if you choose not to use the special version of the optimized math library, you must specify `-QIfdiv-`. This option is the default if you specify either the `-G4` or the `-G6` options.

Avoiding Incorrect Decoding of Certain Instructions (-QI0f)

Some instructions have 2-byte opcodes, the first byte of which contains 0f. In rare cases, the Pentium processor incorrectly decodes these instructions. Specify the `-QI0f` option to avoid the incorrect decoding of these instructions. The work-around implemented in the Intel C/C++ Compiler avoids generating the susceptible instructions.

Controlling Compiler-Generated Code 12

This chapter describes options and features that let you control the outcome of Intel compiler-generated code without interfering with the way your program runs.

- "Specifying Structure Tag Alignments (-Zp)" describes how you can set the alignment of structures and unions from the command line instead of adding a pragma to your source file.
- "Allocation of Zero-initialized Variables (-Qnobss_init)" describes how you can place variables initialized to zero in the `DATA` section instead of allowing the compiler to place them in the `BSS` section.

Specifying Structure Tag Alignments (-Zp)

You can specify an alignment constraint for structures and unions in two ways: place a pack pragma in your source file or enter the alignment option on the command line. Both specifications change structure tag alignment constraints.

Use the `-Zp` option to determine the alignment constraint for structure declarations. Generally, smaller constraints result in smaller data sections while larger constraints support faster execution.

The form of the -Zp option is:

-Zp*number*

number	The alignment constraint indicated by one of the following values:
1	1 byte.
2	2 bytes.
4	4 bytes.
8	8 bytes.
16	16 bytes.

For example, to specify 2 bytes as the alignment constraint for all structures and unions in the file prog1.e, use the following command:

```
prompt> icl -Zp2 prog1.e
```

Allocation of Zero-initialized Variables (-Qnobss_init)

Use the -Qnobss_init option to place any variables that are explicitly initialized with zeros in the _DATA section. By default, variables explicitly initialized with zeros are placed in the _BSS section, but some programs require them to be in the _DATA section.

Support for MMX™ Technology and the Streaming SIMD Extensions 13

The Pentium III Processor and other processors such as the Pentium with MMX technology and Pentium II processors have characteristics that enable the development of advanced multimedia applications. The Streaming SIMD Extensions and MMX technology intrinsics are coding extensions to make use of the processors' multimedia features. This chapter describes how to code with and effectively use these powerful and time saving instructions.

MMX Technology Intrinsics

The benefit of coding with MMX technology intrinsics is that you can use the syntax of C function calls and C variables instead of hardware registers. This frees you from the managing of registers and assembly programming. Further, the compiler optimizes the instruction scheduling so that your executable runs faster.

The intrinsics are based on a new `__m64` data type to represent the specific contents of an `mmx` register.

The `__m64` data type, however, is not a basic ANSI C data type, and therefore you must observe the following usage restrictions:

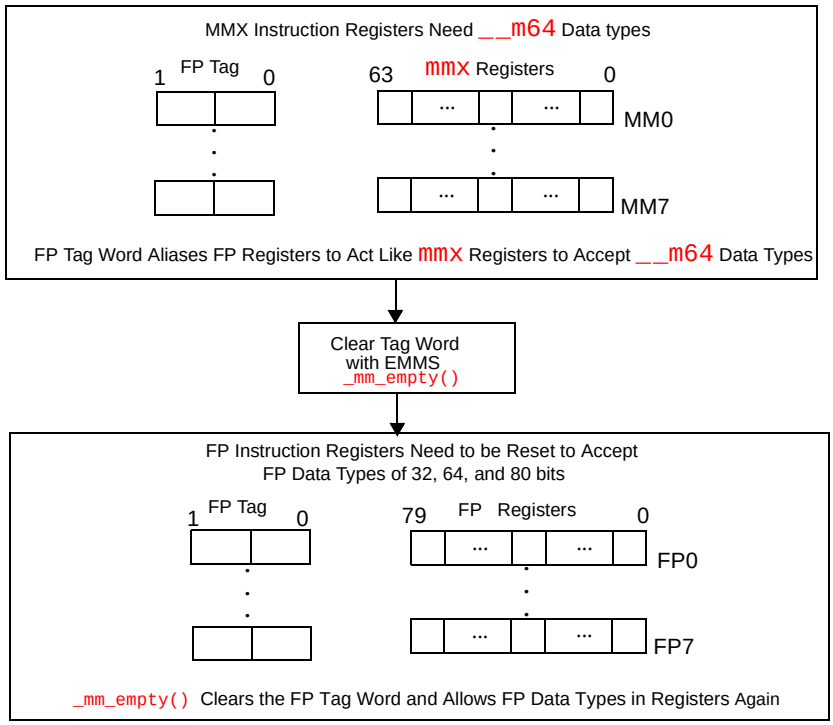
- Use `__m64` data only on the left-hand side of an assignment, as a return value, or as a parameter. You cannot use it with other arithmetic expressions ("`+`", "`>>`", and so on).
- Use `__m64` objects in aggregates, such as unions to access the byte elements and structures; the address of an `__m64` object may be taken.
- Use `__m64` data only with the MMX intrinsics described in this guide.

For complete details of the hardware instructions, see the *Intel Architecture MMX Technology Programmer's Reference Manual*. For descriptions of data types, see the *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual.*

The EMMS Instruction: Why You Need it and When to Use it

Using `EMMS` is like emptying a container to accommodate new content. For instance, MMX instructions automatically enable an FP tag word in the register to enable use of the `__m64` datatype. This resets the FP register set to alias it as the `mmx` register set. To enable the FP register set again, reset the register state with the `EMMS` instruction or via the `_mm_empty()` intrinsic.

Figure 13-1 Why You Need EMMS to Reset After an MMX™ Instruction



CAUTION. *Failure to reset the tag word for FP instructions after using an MMX instruction can result in unexpected execution or poor performance.*

Guidelines for When to Use EMMS

These guidelines help you determine when to use `EMMS`:

- *If next instruction is FP*—Use `_mm_empty()` after an MMX instruction if the next instruction is an FP instruction; for example, before doing calculations on floats, doubles or long doubles.
- *Don't empty when already empty*—If the next instruction uses an MMX register, `_mm_empty()` incurs an operation with no benefit (no-op).
- *Group Instructions*—Use different functions for regions that use FP instructions and those that use MMX instructions. This eliminates needing an EMMS instruction within the body of a critical loop.
- *Runtime initialization*—Use `_mm_empty()` during runtime initialization of `__m64` and FP data types. This ensures resetting the register between data type transitions. See usage coding Example 13-1.

Example 13-1 Correct EMMS Usage In Initialization Code

Incorrect Usage	Correct Usage
`__m64 x = _m_paddd(y, z);` `float f = init();`	`__m64 x = _m_paddd(y, z);` `float f = (_mm_empty(), init());`

Further, you must be aware of all the situations when your code generates an MMX instruction with the Intel C/C++ compiler:

- when using an MMX intrinsic
- when using the Streaming SIMD Extensions (for those intrinsics that use MMX data)
- when using an MMX instruction through inline assembly
- when referencing an `__m64` data type variable

For more documentation on EMMS, visit the following web site:

`http://developer.intel.com`

MMX Technology Intrinsic Groups

The MMX technology intrinsics are grouped into the following sets:

- General Support Intrinsics
- Packed Arithmetic Intrinsics
- Shift Intrinsics
- Logical Intrinsics
- Compare Intrinsics

The syntax is defined for each intrinsics before the definition as follows:

```
data_type intrinsic_name (parameters)                    INST
```

Where,

data_type Is the return data type, which can be either `void`, `int`, or `__m64`. Only the `_mm_empty` intrinsic returns `void`, and only `_mm_cvtsi64_si32` returns an `int`. All other MMX technology intrinsics use the `_m64` data type.

intrinsic_name Is the name of the intrinsic, which behaves like a function that you can use in your C/C++ code instead of inlining the actual instruction. A table in each section provides the intrinsic names, alternate names, and the corresponding instruction. The names listed in the descriptions have a new mnemonic spelling that more easily indicates the functionality. The alternate names are the original intrinsic names, which are provided in case they are the ones that you have previously used.

INST Is the name of the assembly instruction that is used by the intrinsic.Do not use this instruction as part of the syntax in your code. The instruction is placed here to indicate the instruction to which that particular intrinsic corresponds.

General Support Intrinsics

The general support intrinsics are listed in Table 13-1, and are followed by a description of each intrinsic.

Table 13-1 General Support Intrinsics

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_empty	_m_empty	EMMS
_mm_cvtsi32_si64	_m_from_int	MOVD
_mm_cvtsi64_si32	_m_to_int	MOVD
_mm_packs_pi16	_m_packsswb	PACKSSWB
_mm_packs_pi32	_m_packssdw	PACKSSDW
_mm_packs_pu16	_m_packuswb	PACKUSWB
_mm_unpackhi_pi8	_m_punpckhbw	PUNPCKHBW
_mm_unpackhi_pi16	_m_punpckhwd	PUNPCKHWD
_mm_unpackhi_pi32	_m_punpckhdq	PUNPCKHDQ
_mm_unpacklo_pi8	_m_punpcklbw	PUNPCKLBW
_mm_unpacklo_pi16	_m_punpcklwd	PUNPCKLWD
_mm_unpacklo_pi32	_m_punpckldq	PUNPCKLDQ

`void _mm_empty (void)` — EMMS

Empty the multimedia state.
See "The EMMS Instruction: Why You Need it and When to Use it".

`__m64 _mm_cvtsi32_si64 (int i)` — MOVD

Convert the integer object *i* to a 64-bit `__m64` object. The integer value is zero-extended to 64 bits.

`int _mm_cvtsi64_si32 (__m64 m)` — MOVD

Convert the lower 32 bits of the `__m64` object *m* to an integer.

`__m64 _mm_packs_pi16 (__m64 m1, __m64 m2)` — PACKSSWB

Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with signed saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with signed saturation.

`__m64 _mm_packs_pi32 (__m64 m1, __m64 m2)` PACKSSDW

Pack the two 32-bit values from m1 into the lower two 16-bit values of the result with signed saturation, and pack the two 32-bit values from m2 into the upper two 16-bit values of the result with signed saturation.

`__m64 _mm_packs_pu16 (__m64 m1, __m64 m2)` PACKUSWB

Pack the four 16-bit values from m1 into the lower four 8-bit values of the result with unsigned saturation, and pack the four 16-bit values from m2 into the upper four 8-bit values of the result with unsigned saturation.

`__m64 _mm_unpackhi_pi8 (__m64 m1, __m64 m2)` PUNPCKHBW

Interleave the four 8-bit values from the high half of m1 with the four values from the high half of m2 and take the least significant element from m1.

`__m64 _mm_unpackhi_pi16 (__m64 m1, __m64 m2)` PUNPCKHWD

Interleave the two 16-bit values from the high half of *m1* with the two values from the high half of *m2* and take the least significant element from m1.

`__m64 _mm_unpackhi_pi32 (__m64 m1, __m64 m2)` PUNPCKHDQ

Interleave the 32-bit value from the high half of *m1* with the 32-bit value from the high half of *m2* and take the least significant element from m1.

`__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)` PUNPCKLBW

Interleave the four 8-bit values from the low half of *m1* with the four values from the low half of *m2* and take the least significant element from m1.

`__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)` PUNPCKLWD

Interleave the two 16-bit values from the low half of *m1* with the two values from the low half of *m2* and take the least significant element from m1.

`__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)` PUNPCKLDQ

Interleave the 32-bit value from the low half of *m1* with the 32-bit value from the low half of *m2* and take the least significant element from m1.

Packed Arithmetic Intrinsics

The packed arithmetic intrinsics are listed in Table , and are followed by a description of each intrinsic.

Table 13-2 Packed Arithmetic Intrinsics

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_add_pi8	_m_paddb	PADDB
_mm_add_pi16	_m_paddw	PADDW
_mm_add_pi32	_m_paddd	PADDD
_mm_adds_pi8	_m_paddsb	PADDSB
_mm_adds_pi16	_m_paddsw	PADDSW
_mm_adds_pu8	_m_paddusb	PADDUSB
_mm_adds_pu16	_m_paddusw	PADDUSW
_mm_sub_pi8	_m_psubb	PSUBB
_mm_sub_pi16	_m_psubw	PSUBW
_mm_sub_pi32	_m_psubd	PSUBD
_mm_subs_pi8	_m_psubsb	PSUBSB
_mm_subs_pi16	_m_psubsw	PSUBSW
_mm_subs_pu8	_m_psubusb	PSUBUSB
_mm_subs_pu16	_m_psubusw	PSUBUSW
_mm_madd_pi16	_m_pmaddwd	PMADDWD
_mm_mulhi_pi16	_m_pmulhw	PMULHW
_mm_mullo_pi16	_m_pmullw	PMULLW

`__m64 _mm_add_pi8 (__m64 m1, __m64 m2)` PADDB

Add the eight 8-bit values in *m1* to the eight 8-bit values in *m2*.

`__m64 _mm_add_pi16 (__m64 m1, __m64 m2)` PADDW

Add the four 16-bit values in *m1* to the four 16-bit values in *m2*.

`__m64 _mm_add_pi32 (__m64 m1, __m64 m2)` PADDD

Add the two 32-bit values in *m1* to the two 32-bit values in *m2*.

`__m64 _mm_adds_pi8 (__m64 m1, __m64 m2)` PADDSB

Add the eight signed 8-bit values in *m1* to the eight signed 8-bit values in *m2* and saturate.

`__m64 _mm_adds_pi16 (__m64 m1, __m64 m2)` PADDSW

Add the four signed 16-bit values in *m1* to the four signed 16-bit values in *m2* and saturate.

`__m64 _mm_adds_pu8 (__m64 m1, __m64 m2)` PADDUSB

Add the eight unsigned 8-bit values in *m1* to the eight unsigned 8-bit values in *m2* and saturate.

`__m64 _mm_adds_pu16 (__m64 m1, __m64 m2)` PADDUSW

Add the four unsigned 16-bit values in *m1* to the four unsigned 16-bit values in *m2* and saturate.

`__m64 _mm_sub_pi8 (__m64 m1, __m64 m2)` PSUBB

Subtract the eight 8-bit values in *m2* from the eight 8-bit values in *m1*.

`__m64 _mm_sub_pi16 (__m64 m1, __m64 m2)` PSUBW

Subtract the four 16-bit values in *m2* from the four 16-bit values in *m1*.

`__m64 _mm_sub_pi32 (__m64 m1, __m64 m2)` PSUBD

Subtract the two 32-bit values in *m2* from the two 32-bit values in *m1*.

`__m64 _mm_subs_pi8 (__m64 m1, __m64 m2)` PSUBSB

Subtract the eight signed 8-bit values in *m2* from the eight signed 8-bit values in *m1* and saturate.

`__m64 _mm_subs_pi16 (__m64 m1, __m64 m2)` PSUBSW

Subtract the four signed 16-bit values in *m2* from the four signed 16-bit values in *m1* and saturate.

`__m64 _mm_subs_pu8 (__m64 m1, __m64 m2)` PSUBUSB

Subtract the eight unsigned 8-bit values in *m2* from the eight unsigned 8-bit values in *m1* and saturate.

`__m64 _mm_subs_pu16 (__m64 m1, __m64 m2)` PSUBUSW

Subtract the four unsigned 16-bit values in *m2* from the four unsigned 16-bit values in *m1* and saturate.

`__m64 _mm_madd_pi16 (__m64 m1, __m64 m2)` PMADDWD

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* producing four 32-bit intermediate results, which are then summed by pairs to produce two 32-bit results.

`__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)` PMULHW

Multiply four signed 16-bit values in *m1* by four signed 16-bit values in *m2* and produce the high 16 bits of the four results.

`__m64 _mm_mullo_pi16 (__m64 m1, __m64 m2)` PMULLW

Multiply four 16-bit values in *m1* by four 16-bit values in *m2* and produce the low 16 bits of the four results.

Shift Intrinsics

The shift intrinsics are listed in Table , and are followed by a description of each intrinsic.

Table 13-3 Shift Intrinsics

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_sll_pi16	_m_psllw	PSLLW
_mm_slli_pi16	_m_psllwi	PSLLW
_mm_sll_pi32	_m_pslld	PSLLD
_mm_slli_pi32	_m_pslldi	PSLLD

Table 13-3 Shift Intrinsics (Continued)

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_sll_si64	_m_psllq	PSLLQ
_mm_slli_si64	_m_psllqi	PSLLQ
_mm_sra_pi16	_m_psraw	PSRAW
_mm_srai_pi16	_m_psrawi	PSRAW
_mm_sra_pi32	_m_psrad	PSRAD
_mm_srai_pi32	_m_psradi	PSRADI
_mm_srl_pi16	_m_psrlw	PSRLW
_mm_srli_pi16	_m_psrlwi	PSRLW
_mm_srl_pi32	_m_psrld	PSRLD
_mm_srli_pi32	_m_psrldi	PSRLD
_mm_srl_si64	_m_psrlq	PSRLQ
_mm_srli_si64	_m_psrlqi	PSRLQ

`__m64 _mm_sll_pi16 (__m64 m, __m64 count)` PSLLW

Shift four 16-bit values in *m* left the amount specified by *count* while shifting in zeros.

`__m64 _mm_slli_pi16 (__m64 m, int count)` PSLLW

Shift four 16-bit values in *m* left the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

`__m64 _mm_sll_pi32 (__m64 m, __m64 count)` PSLLD

Shift two 32-bit values in *m* left the amount specified by *count* while shifting in zeros.

`__m64 _mm_slli_pi32 (__m64 m, int count)` PSLLD

Shift two 32-bit values in *m* left the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

`__m64 _mm_sll_si64 (__m64 m, __m64 count)` PSLLQ

Shift the 64-bit value in *m* left the amount specified by *count* while shifting in zeros.

`__m64 _mm_slli_si64 (__m64 m, int count)` PSLLQ

Shift the 64-bit value in *m* left the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

`__m64 _mm_sra_pi16 (__m64 m, __m64 count)` PSRAW

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in the sign bit.

`__m64 _mm_srai_pi16 (__m64 m, int count)` PSRAW

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in the sign bit. For the best performance, *count* should be a constant.

`__m64 _mm_sra_pi32 (__m64 m, __m64 count)` PSRAD

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in the sign bit.

`__m64 _mm_srai_pi32 (__m64 m, int count)` PSRADI

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in the sign bit. For the best performance, *count* should be a constant.

`__m64 _mm_srl_pi16 (__m64 m, __m64 count)` PSRLW

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in zeros.

`__m64 _mm_srli_pi16 (__m64 m, int count)` PSRLW

Shift four 16-bit values in *m* right the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

`__m64 _mm_srl_pi32 (__m64 m, __m64 count)` PSRLD

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in zeros.

`__m64 _mm_srli_pi32 (__m64 m, int count)` PSRLD

Shift two 32-bit values in *m* right the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

`__m64 _mm_srl_si64 (__m64 m, __m64 count)` PSRLQ

Shift the 64-bit value in *m* right the amount specified by *count* while shifting in zeros.

`__m64 _mm_srli_si64 (__m64 m, int count)` PSRLQ

Shift the 64-bit value in *m* right the amount specified by *count* while shifting in zeros. For the best performance, *count* should be a constant.

Logical Intrinsics

The logical intrinsics are listed in Table 13-4, and are followed by a description of each intrinsic

Table 13-4 Logical Intrinsics

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_and_si64	_m_pand	PAND
_mm_andnot_si64	_m_pandn	PANDN
_mm_or_si64	_m_por	POR
_mm_xor_si64	_m_pxor	PXOR

`__m64 _mm_and_si64 (__m64 m1, __m64 m2)` PAND

Perform a bitwise AND of the 64-bit value in *m1* with the 64-bit value in *m2*.

`__m64 _mm_andnot_si64 (__m64 m1, __m64 m2)` PANDN

Perform a logical NOT on the 64-bit value in *m1* and use the result in a bitwise AND with the 64-bit value in *m2*.

`__m64 _mm_or_si64 (__m64 m1, __m64 m2)` POR

Perform a bitwise OR of the 64-bit value in *m1* with the 64-bit value in *m2*.

`__m64 _mm_xor_si64 (__m64 m1, __m64 m2)` PXOR

Perform a bitwise XOR of the 64-bit value in *m1* with the 64-bit value in *m2*.

Compare Intrinsics

The compare intrinsics are listed in Table 13-4, and are followed by a description of each intrinsic.

Table 13-5 Compare Intrinsics

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_cmpeq_pi8	_m_pcmpeqb	PCMPEQB
_mm_cmpeq_pi16	_m_pcmpeqw	PCMPEQW
_mm_cmpeq_pi32	_m_pcmpeqd	PCMPEQD
_mm_cmpgt_pi8	_m_pcmpgtb	PCMPGTB
_mm_cmpgt_pi16	_m_pcmpgtw	PCMPGTW
_mm_cmpgt_pi32	_m_pcmpgtd	PCMPGTD

`__m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)` PCMPEQB

If the respective 8-bit values in *m1* are equal to the respective 8-bit values in *m2* set the respective 8-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)` PCMPEQW

If the respective 16-bit values in *m1* are equal to the respective 16-bit values in *m2* set the respective 16-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)` PCMPEQD

If the respective 32-bit values in *m1* are equal to the respective 32-bit values in *m2* set the respective 32-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)` PCMPGTB

If the respective 8-bit values in *m1* are greater than the respective 8-bit values in *m2* set the respective 8-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _mm_cmpgt_pi16 (__m64 m1, __m64 m2)` PCMPGTW

If the respective 16-bit values in *m1* are greater than the respective 16-bit values in *m2* set the respective 16-bit resulting values to all ones, otherwise set them to all zeros.

`__m64 _mm_cmpgt_pi32 (__m64 m1, __m64__m64 m2)` PCMPGTD

If the respective 32-bit values in *m1* are greater than the respective 32-bit values in *m2* set the respective 32-bit resulting values to all ones, otherwise set them all to zeros.

Processor-Dispatching Support

With `__declspec(cpu_specific)` and `__declspec(cpu_dispatch)`, you can direct the compiler to generate code that automatically determines the processor on which it is running, and selects the appropriate implementation of a function. This enables you, for example, to write code that takes advantage of MMX instructions when executing on a processor that has them, but that also executes correctly on older processors that do not have the MMX instructions. The syntax of these extended-attributes is as follows:

`cpu_specific(`*`cpuid`*`)`
`cpu_dispatch(`*`cpuid-list`*`)`

where *cpuid* is one of:

```
generic
pentium
pentium_pro
pentium_mmx
pentium_ii
pentium_iii
pentium_iii_no_xmm_regs
```

and *cpuid-list* is one of:

cpuid
cpuid-list , cpuid

The names of the *cpuid* are not case sensitive. You can specify these attributes only for function definitions. The body of a function declared with `__declspec(cpu_dispatch)` must be empty, and is referred to as a stub.

If a function *f* is defined as `__declspec(cpu_specific(`*p*`))`, then a `cpu_dispatch` stub must appear for *f* somewhere in the program, and *p* must be in the *cpuid-list* of that stub; otherwise, that `cpu_specific` definition can never be called and no error will be reported for this condition.

If the `cpu_dispatch` stub for a function *f* contains the *cpuid p*, then a `cpu_specific` definition of *f* with *cpuid p* must appear somewhere in the program; otherwise an unresolved external error is reported. A `cpu_specific` function definition need not appear in the same translation unit as the corresponding `cpu_dispatch` stub, unless the `cpu_specific` function is declared `static`. The `inline` attribute is disabled for all `cpu_specific` and `cpu_dispatch` functions.

When a `cpu_dispatch` stub is compiled, its body is filled in with code that determines the processor on which the program is running, then dispatches to the "best" `cpu_specific` implementation available (as defined by the *cpuid-list*) that will run on that processor. When a `cpu_specific` function is compiled, optimizations and code generation options appropriate to the specified processor are used regardless of command-line option settings.

Here is an example of how these features can be used:

```
#include <mmintrin.h>

/* array_sum(r, a, b, l) adds two arrays of unsigned
short (a and b),of length l, and stores the result in r.
*/

__declspec(cpu_specific(Pentium))
void array_sum(unsigned short *result,
unsigned short const *a,
```

```
unsigned short const *b,
size_t length)

{
/* The implementation specific to the Pentium processor
uses no special architectural features. */

for (; length > 0; length--)
   *result++ = *a++ + *b++;
}
__declspec(cpu_specific(Pentium_MMX))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
/* The implementation for a Pentium processor with MMX
technology uses uses an MMX instruction intrinsic to add
four elements at a time, to reduce the number of loop
iterations by 3/4. */

__m64 *mmx_result = (__m64 *)result;
__m64 const *mmx_a = (__m64 const *)a;
__m64 const *mmx_b = (__m64 const *)b;
for (; length > 3; length -= 4)
   *mmx_result++ = _m_paddw(*mmx_a++, *mmx_b++);

/* If the size of all arrays passed to this routine is
known to be a multiple of four, the following code
(which takes care of excess elements) is not necessary.
*/

result = (unsigned short *)mmx_result;
a = (unsigned short const *)mmx_a;
b = (unsigned short const *)mmx_b;
for (; length > 0; length--)
   *result++ = *a++ + *b++;
```

```
}

__declspec(cpu_dispatch(Pentium, Pentium_MMX))
void array_sum(unsigned short *result,
unsigned short const *a,
unsigned short const *b,
size_t length)
{
/* An empty function body, which informs the compiler
that it should generate a dispatch function for the
CPU-specific implementations listed in the cpu_dispatch
clause. */
}
```

Streaming SIMD Extensions Intrinsics

This section describes the C/C++ language-level features supporting the Streaming SIMD Extensions in the Intel C/C++ Compiler. The section explains the following features of the intrinsics:

- the intrinsics API
- `__m128` datatype
- data alignment support
- assembly language support

This section provides the complete list of intrinsics for the Streaming SIMD Extensions.

The Intrinsics API

The intrinsics for the Streaming SIMD Extensions allow you to access the functionality provided by the new instructions without actually using assembly language. Just like the intrinsics for the integer MMX instructions, for each computational and data manipulation instruction in the new instruction set, there is a corresponding C intrinsic that implements it directly. A new C data type, representing the 128-bit registers, is used as

the operand to these intrinsic functions. The intrinsics allow you to specify the underlying implementation (instruction selection) of an algorithm yet leave instruction scheduling and register allocation to the compiler.

The __m128 Data Type

The __m128 data type is used to represent the contents of an xmm register, which is either four packed single-precision floating-point values or one scalar single-precision number. The __m128 data type is not a basic ANSI C data type and therefore some restrictions are placed on its usage:

- Use __m128 only on the left-hand side of an assignment, as a return value, or as a parameter. Do not use it in other arithmetic expressions such as "+" and ">>".
- Do not initialize __m128 with literals; there is no way to express 128-bit constants.
- Use __m128 objects in aggregates, such as unions (for example, to access the float elements) and structures. The address of an __m128 object may be taken.
- Use __m128 data only with the intrinsics described in this user's guide.

The compiler aligns __m128 local data to 16B boundaries on the stack. Global __m128 data is also 16B-aligned. (To align float arrays, you can use the alignment declspec described in the following section.) Because the new instruction set treats the Streaming SIMD Extensions registers in the same way whether you are using packed or scalar data, there is no __m32 datatype to represent scalar data as you might expect. For scalar operations, you should use the __m128 objects and the "scalar" forms of the intrinsics; the compiler and the processor implement these operations with 32-bit memory references.

Streaming SIMD Extensions Intrinsic Conventions

The list of intrinsics for the Streaming SIMD Extensions uses a number of conventions described in the following paragraphs.

The suffixes ps and ss are used to denote "packed single" and "scalar single" precision operations. The packed floats are represented in right-to-left order, with the lowest word (right-most) being used for scalar operations: [z, y, x, w]. To explain how memory storage reflects this, consider the following example. The operation

```
float a[4] = { 1.0, 2.0, 3.0, 4.0 };
__m128 t = _mm_load_ps(a);
```

produces the same result as follows:

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

In other words,

```
t = [ 4.0, 3.0, 2.0, 1.0 ]
```

where the “scalar” element is `1.0`.

Floating Point Intrinsics

The following sections list the floating-point intrinsics broken into groups by the nature of the operation. Each intrinsic entry has an informal pseudo-code and it is followed with a corresponding instruction name in upper case letters; for example, `ADDSS` is the name of the first instruction listed in this section, which corresponds to the intrinsic for `__m128 _mm_add_ss(__m128 a, __m128 b)`.

The variable `r` is generally used for the intrinsic’s return value. A number appended to a variable name indicates the element of a packed object. For example, `r0` is the lowest word of `r`. Some intrinsics are “composites” because they require more than one instruction to implement them.

You should be familiar with the hardware features provided by the Streaming SIMD Extensions when writing programs with the intrinsics. The following are three important issues to keep in mind:

- Certain intrinsics, such as `_mm_loadr_ps` and `_mm_cmpgt_ss`, are not directly supported by the instruction set. While these intrinsics are convenient programming aids, be mindful that they might consist of more than one machine-language instruction.
- Floating-point data loaded or stored as `__m128` objects must be generally 16-byte-aligned.
- Some intrinsics require that their argument be immediates, that is, constant integers (literals), due to the nature of the instruction.

- The result of arithmetic operations acting on two NaN (Not a Number) arguments is undefined. Therefore, FP operations using NaN arguments will not match the expected behavior of the corresponding assembly instructions.

Arithmetic Operations

`__m128 _mm_add_ss(__m128 a, __m128 b)` ADDSS

Adds the lower SP FP (single-precision, floating-point) values of *a* and *b*; the upper 3 SP FP values are passed through from *a*.

```
r0 := a0 + b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_add_ps(__m128 a, __m128 b)` ADDPS

Adds the four SP FP values of *a* and *b*.

```
r0 := a0 + b0
r1 := a1 + b1
r2 := a2 + b2
r3 := a3 + b3
```

`__m128 _mm_sub_ss(__m128 a, __m128 b)` SUBSS

Subtracts the lower SP FP values of *a* and *b*. The upper 3 SP FP values are passed through from *a*.

```
r0 := a0 - b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_sub_ps(__m128 a, __m128 b)` SUBPS

Subtracts the four SP FP values of *a* and *b*.

```
r0 := a0 - b0
r1 := a1 – b1
r2 := a2 – b2
r3 := a3 – b3
```

`__m128 _mm_mul_ss(__m128 a, __m128 b)` MULSS

Multiplies the lower SP FP values of *a* and *b*; the upper 3 SP FP values are passed through from *a*.

```
r0 := a0 * b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_mul_ps(__m128 a, __m128 b)` MULPS

Multiplies the four SP FP values of *a* and *b*.

```
r0 := a0 * b0
r1 := a1 * b1
r2 := a2 * b2
r3 := a3 * b3
```

`__m128 _mm_div_ss(__m128 a, __m128 b)` DIVSS

Divides the lower SP FP values of *a* and *b*; the upper 3 SP FP values are passed through from *a*.

```
r0 := a0 / b0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_div_ps(__m128 a, __m128 b)` DIVPS

Divides the four SP FP values of *a* and *b*.

```
r0 := a0 / b0
r1 := a1 / b1
r2 := a2 / b2
r3 := a3 / b3
```

`__m128 _mm_sqrt_ss(__m128 a)` SQRTSS

Computes the square root of the lower SP FP value of *a*; the upper 3 SP FP values are passed through.

```
r0 := sqrt(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_sqrt_ps(__m128 a)` SQRTPS

Computes the square roots of the four SP FP values of *a*.

```
r0 := sqrt(a0)
```

```
r1 := sqrt(a1)
r2 := sqrt(a2)
r3 := sqrt(a3)
```

`__m128 _mm_rcp_ss(__m128 a)` RCPSS

Computes the approximation of the reciprocal of the lower SP FP value of *a*; the upper 3 SP FP values are passed through.

```
r0 := recip(a0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_rcp_ps(__m128 a)` RCPPS

Computes the approximations of reciprocals of the four SP FP values of *a*.

```
r0 := recip(a0)
r1 := recip(a1)
r2 := recip(a2)
r3 := recip(a3)
```

`__m128 _mm_rsqrt_ss(__m128 a)` RSQRTSS

Computes the approximation of the reciprocal of the square root of the lower SP FP value of *a*; the upper 3 SP FP values are passed through.

```
r0 := recip(sqrt(a0))
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_rsqrt_ps(__m128 a)` RSQRTPS

Computes the approximations of the reciprocals of the square roots of the four SP FP values of *a*.

```
r0 := recip(sqrt(a0))
r1 := recip(sqrt(a1))
r2 := recip(sqrt(a2))
r3 := recip(sqrt(a3))
```

`__m128 _mm_min_ss(__m128 a, __m128 b)` MINSS

Computes the minimum of the lower SP FP values of *a* and *b*; the upper 3 SP FP values are passed through from *a*.

```
r0 := min(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_min_ps(__m128 a, __m128 b)` MINPS

Computes the minimums of the four SP FP values of *a* and *b*.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

`__m128 _mm_max_ss(__m128 a, __m128 b)` MAXSS

Computes the maximum of the lower SP FP values of *a* and *b*; the upper 3 SP FP values are passed through from *a*.

```
r0 := max(a0, b0)
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_max_ps(__m128 a, __m128 b)` MAXPS

Computes the maximums of the four SP FP values of *a* and *b*.

```
r0 := max(a0, b0)
r1 := max(a1, b1)
r2 := max(a2, b2)
r3 := max(a3, b3)
```

Logical Operations

`__m128 _mm_and_ps(__m128 a, __m128 b)` ANDPS

Computes the bitwise And of the four SP FP values of *a* and *b*.

```
r0 := a0 & b0
r1 := a1 & b1
r2 := a2 & b2
r3 := a3 & b3
```

`__m128 _mm_andnot_ps(__m128 a, __m128 b)` ANDNPS

Computes the bitwise AND-NOT of the four SP FP values of *a* and *b*.

```
r0 := ~a0 & b0
r1 := ~a1 & b1
r2 := ~a2 & b2
r3 := ~a3 & b3
```

`__m128 _mm_or_ps(__m128 a, __m128 b)` ORPS

Computes the bitwise OR of the four SP FP values of *a* and *b*.

```
r0 := a0 | b0
r1 := a1 | b1
r2 := a2 | b2
r3 := a3 | b3
```

`__m128 _mm_xor_ps(__m128 a, __m128 b)` XORPS

Computes bitwise EXOR (exclusive-or) of the four SP FP values of *a* and *b*.

```
r0 := a0 ^ b0
r1 := a1 ^ b1
r2 := a2 ^ b2
r3 := a3 ^ b3
```

Comparisons

Each comparison intrinsic performs a comparison of *a* and *b*. For the packed form, the four SP FP values of *a* and *b* are compared, and a 128-bit mask is returned. For the scalar form, the lower SP FP values of *a* and *b* are compared, and a 32-bit mask is returned; the upper three SP FP values are passed through from *a*. The mask is set to `0xffffffff` for each element where the comparison is true and `0x0` where the comparison is false.

The superscript 'r' on the instruction indicates operands are reversed in the instruction implementation.

`__m128 _mm_cmpeq_ss(__m128 a, __m128 b)` CMPEQSS

Compare for equality.

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
```

```
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpeq_ps(__m128 a, __m128 b)` CMPEQPS

Compare for equality.

```
r0 := (a0 == b0) ? 0xffffffff : 0x0
r1 := (a1 == b1) ? 0xffffffff : 0x0
r2 := (a2 == b2) ? 0xffffffff : 0x0
r3 := (a3 == b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmplt_ss(__m128 a, __m128 b)` CMPLTSS

Compare for less-than.

```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmplt_ps(__m128 a, __m128 b)` CMPLTPS

Compare for less-than.

```
r0 := (a0 < b0) ? 0xffffffff : 0x0
r1 := (a1 < b1) ? 0xffffffff : 0x0
r2 := (a2 < b2) ? 0xffffffff : 0x0
r3 := (a3 < b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmple_ss(__m128 a, __m128 b)` CMPLESS

Compare for less-than-or-equal.

```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmple_ps(__m128 a, __m128 b)` CMPLEPS

Compare for less-than-or-equal.

```
r0 := (a0 <= b0) ? 0xffffffff : 0x0
r1 := (a1 <= b1) ? 0xffffffff : 0x0
r2 := (a2 <= b2) ? 0xffffffff : 0x0
r3 := (a3 <= b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpgt_ss(__m128 a, __m128 b)` CMPLTSS[r]

Compare for greater-than.

```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpgt_ps(__m128 a, __m128 b)` CMPLTPS[r]

Compare for greater-than.

```
r0 := (a0 > b0) ? 0xffffffff : 0x0
r1 := (a1 > b1) ? 0xffffffff : 0x0
r2 := (a2 > b2) ? 0xffffffff : 0x0
r3 := (a3 > b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpge_ss(__m128 a, __m128 b)` CMPLESS[r]

Compare for greater-than-or-equal.

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpge_ps(__m128 a, __m128 b)` CMPLEPS[r]

Compare for greater-than-or-equal.

```
r0 := (a0 >= b0) ? 0xffffffff : 0x0
r1 := (a1 >= b1) ? 0xffffffff : 0x0
r2 := (a2 >= b2) ? 0xffffffff : 0x0
r3 := (a3 >= b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpneq_ss(__m128 a, __m128 b)` CMPNEQSS

Compare for inequality.

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpneq_ps(__m128 a, __m128 b)` CMPNEQPS

Compare for inequality.

```
r0 := (a0 != b0) ? 0xffffffff : 0x0
r1 := (a1 != b1) ? 0xffffffff : 0x0
```

```
r2 := (a2 != b2) ? 0xffffffff : 0x0
r3 := (a3 != b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)` CMPNLTSS

Compare for not-less-than.

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpnlt_ps(__m128 a, __m128 b)` CMPNLTPS

Compare for not-less-than.

```
r0 := !(a0 < b0) ? 0xffffffff : 0x0
r1 := !(a1 < b1) ? 0xffffffff : 0x0
r2 := !(a2 < b2) ? 0xffffffff : 0x0
r3 := !(a3 < b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpnle_ss(__m128 a, __m128 b)` CMPNLESS

Compare for not-less-than-or-equal.

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpnle_ps(__m128 a, __m128 b)` CMPNLEPS

Compare for not-less-than-or-equal.

```
r0 := !(a0 <= b0) ? 0xffffffff : 0x0
r1 := !(a1 <= b1) ? 0xffffffff : 0x0
r2 := !(a2 <= b2) ? 0xffffffff : 0x0
r3 := !(a3 <= b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpngt_ss(__m128 a, __m128 b)` CMPNLTSS[r]

Compare for not-greater-than.

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpngt_ps(__m128 a, __m128 b)` CMPNLTPS[r]

Compare for not-greater-than.

```
r0 := !(a0 > b0) ? 0xffffffff : 0x0
r1 := !(a1 > b1) ? 0xffffffff : 0x0
r2 := !(a2 > b2) ? 0xffffffff : 0x0
r3 := !(a3 > b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpnge_ss(__m128 a, __m128 b)` CMPNLESS[r]

Compare for not-greater-than-or-equal.

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpnge_ps(__m128 a, __m128 b)` CMPNLEPS[r]

Compare for not-greater-than-or-equal.

```
r0 := !(a0 >= b0) ? 0xffffffff : 0x0
r1 := !(a1 >= b1) ? 0xffffffff : 0x0
r2 := !(a2 >= b2) ? 0xffffffff : 0x0
r3 := !(a3 >= b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpord_ss(__m128 a, __m128 b)` CMPORDSS

Compare for ordered.

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpord_ps(__m128 a, __m128 b)` CMPORDPS

Compare for ordered.

```
r0 := (a0 ord? b0) ? 0xffffffff : 0x0
r1 := (a1 ord? b1) ? 0xffffffff : 0x0
r2 := (a2 ord? b2) ? 0xffffffff : 0x0
r3 := (a3 ord? b3) ? 0xffffffff : 0x0
```

`__m128 _mm_cmpunord_ss(__m128 a, __m128 b)` CMPUNORDSS

Compare for unordered.

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cmpunord_ps(__m128 a, __m128 b)` CMPUNORDPS

Compare for unordered.

```
r0 := (a0 unord? b0) ? 0xffffffff : 0x0
r1 := (a1 unord? b1) ? 0xffffffff : 0x0
r2 := (a2 unord? b2) ? 0xffffffff : 0x0
r3 := (a3 unord? b3) ? 0xffffffff : 0x0
```

`int _mm_comieq_ss (__m128 a, __m128 b)` COMISS

Compares the lower SP FP value of *a* and *b* for *a* equal to *b*. If *a* and *b* are equal, `1` is returned. Otherwise `0` is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

`int _mm_comilt_ss (__m128 a, __m128 b)` COMISS

Compares the lower SP FP value of *a* and *b* for *a* less than *b*. If *a* is less than *b*, `1` is returned. Otherwise `0` is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

`int _mm_comile_ss (__m128 a, __m128 b)` COMISS

Compares the lower SP FP value of *a* and *b* for *a* less than or equal to *b*. If *a* is less than or equal to *b*, `1` is returned. Otherwise `0` is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

`int _mm_comigt_ss (__m128 a, __m128 b)` COMISS

Compares the lower SP FP value of *a* and *b* for *a* greater than *b*. If *a* is greater than *b* are equal, `1` is returned. Otherwise `0` is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

`int _mm_comige_ss (__m128 a, __m128 b)` COMISS

Compares the lower SP FP value of *a* and *b* for *a* greater than or equal to *b*. If *a* is greater than or equal to *b*, `1` is returned. Otherwise `0` is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

`int _mm_comineq_ss (__m128 a, __m128 b)` COMISS

Compares the lower SP FP value of *a* and *b* for *a* not equal to *b*. If *a* and *b* are not equal, `1` is returned. Otherwise `0` is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

`int _mm_ucomieq_ss (__m128 a, __m128 b)` UCOMISS

Compares the lower SP FP value of *a* and *b* for *a* equal to *b*. If *a* and *b* are equal, `1` is returned. Otherwise `0` is returned.

```
r := (a0 == b0) ? 0x1 : 0x0
```

`int _mm_ucomilt_ss (__m128 a, __m128 b)` UCOMISS

Compares the lower SP FP value of *a* and *b* for *a* less than *b*. If *a* is less than *b*, `1` is returned. Otherwise `0` is returned.

```
r := (a0 < b0) ? 0x1 : 0x0
```

`int _mm_ucomile_ss (__m128 a, __m128 b)` UCOMISS

Compares the lower SP FP value of *a* and *b* for *a* less than or equal to *b*. If *a* is less than or equal to *b*, `1` is returned. Otherwise `0` is returned.

```
r := (a0 <= b0) ? 0x1 : 0x0
```

`int _mm_ucomigt_ss (__m128 a, __m128 b)` UCOMISS

Compares the lower SP FP value of *a* and *b* for *a* greater than *b*. If *a* is greater than *b* are equal, `1` is returned. Otherwise `0` is returned.

```
r := (a0 > b0) ? 0x1 : 0x0
```

`int _mm_ucomige_ss (__m128 a, __m128 b)` UCOMISS

Compares the lower SP FP value of *a* and *b* for *a* greater than or equal to *b*. If *a* is greater than or equal to *b*, `1` is returned. Otherwise `0` is returned.

```
r := (a0 >= b0) ? 0x1 : 0x0
```

`int _mm_ucomineq_ss (__m128 a, __m128 b)` UCOMISS

Compares the lower SP FP value of *a* and *b* for *a* not equal to *b*. If *a* and *b* are not equal, `1` is returned. Otherwise `0` is returned.

```
r := (a0 != b0) ? 0x1 : 0x0
```

Conversion Operations

The conversions operations are listed in Table 13-6, followed by a description of each intrinsic with the most recent mnemonic naming convention. The alternate name is provided in case you have used these intrinsics before.

Table 13-6 Conversion Operations

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_cvtss_si32	_mm_cvt_ss2si	CVTSS2SI
_mm_cvtps_pi32	_mm_cvt_ps2pi	CVTPS2PI
_mm_cvttss_si32	_mm_cvtt_ss2si	CVTTSS2SI
_mm_cvttps_pi32	_mm_cvtt_ps2pi	CVTTPS2PI
_mm_cvtsi32_ss	_mm_cvt_si2ss	CVTSI2SS
_mm_cvtpi32_ps	_mm_cvt_pi2ps	CVTTPS2PI

`int _mm_cvtss_si32(__m128 a)` CVTSS2SI

Convert the lower SP FP value of *a* to a 32-bit integer according to the current rounding mode.

```
r := (int)a0
```

`__m64 _mm_cvtps_pi32(__m128 a)` CVTPS2PI

Convert the two lower SP FP values of *a* to two 32-bit integers according to the current rounding mode, returning the integers in packed form.

```
r0 := (int)a0
```

```
r1 := (int)a1
```

`int _mm_cvttss_si32(__m128 a)` CVTTSS2SI

Convert the lower SP FP value of *a* to a 32-bit integer with truncation.

```
r := (int)a0
```

`__m64 _mm_cvttps_pi32(__m128 a)` CVTTPS2PI

Convert the two lower SP FP values of *a* to two 32-bit integer with truncation, returning the integers in packed form.

```
r0 := (int)a0
r1 := (int)a1
```

`__m128 _mm_cvtsi32_ss(__m128 a, int b)` CVTSI2SS

Convert the 32-bit integer value *b* to an SP FP value; the upper three SP FP values are passed through from *a*.

```
r0 := (float)b
r1 := a1 ; r2 := a2 ; r3 := a3
```

`__m128 _mm_cvtpi32_ps(__m128 a, __m64 b)` CVTPI2PS

Convert the two 32-bit integer values in packed form in *b* to two SP FP values; the upper two SP FP values are passed through from *a*.

```
r0 := (float)b0
r1 := (float)b1
r2 := a2
r3 := a3
```

`__m128 _mm_cvtpi16_ps(__m64 a)`(composite)

Convert the four 16-bit signed integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

`__m128 _mm_cvtpu16_ps(__m64 a)`(composite)

Convert the four 16-bit unsigned integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

`__m128 _mm_cvtpi8_ps(__m64 a)`(composite)

Convert the lower four 8-bit signed integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

`__m128 _mm_cvtpu8_ps(__m64 a)` (composite)

Convert the lower four 8-bit unsigned integer values in a to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)a2
r3 := (float)a3
```

`__m128 _mm_cvtpi32x2_ps(__m64 a, __m64 b)` (composite)

Convert the two 32-bit signed integer values in a and the two 32-bit signed integer values in b to four single precision FP values.

```
r0 := (float)a0
r1 := (float)a1
r2 := (float)b0
r3 := (float)b1
```

`__m64 _mm_cvtps_pi16(__m128 a)` (composite)

Convert the four single precision FP values in a to four signed 16-bit integer values.

```
r0 := (short)a0
r1 := (short)a1
```

```
r2 := (short)a2
r3 := (short)a3
```

`__m64 _mm_cvtps_pi8(__m128 a)` (composite)

Convert the four single precision FP values in `a` to the lower four signed 8-bit integer values of the result.

```
r0 := (char)a0
r1 := (char)a1
r2 := (char)a2
r3 := (char)a3
```

Miscellaneous

`__m128 _mm_shuffle_ps(__m128 a, __m128 b, int i) SHUFPS`

Selects four specific SP FP values from *a* and *b*, based on the mask *i*. The mask must be an immediate. See "Macro Function for Shuffle" in the end of this section for a description of the shuffle semantics.

`__m128 _mm_unpackhi_ps(__m128 a, __m128 b)` `UNPCKHPS`

Selects and interleaves the upper two SP FP values from *a* and *b*.

```
r0 := a2
r1 := b2
r2 := a3
r3 := b3
```

`__m128 _mm_unpacklo_ps(__m128 a, __m128 b)` `UNPCKLPS`

Selects and interleaves the lower two SP FP values from *a* and *b*.

```
r0 := a0
r1 := b0
r2 := a1
r3 := b1
```

`__m128 _mm_loadh_pi(__m128 a, __m64 *p) MOVHPS reg, mem`

Sets the upper two SP FP values with 64 bits of data loaded from the address *p*; the lower two values are passed through from *a*.

```
r0 := a0
r1 := a1
r2 := *p0
r3 := *p1
```

```
void _mm_storeh_pi(__m64 *p, __m128 a)      MOVHPS mem, reg
```

Stores the upper two SP FP values of *a* to the address *p*.

```
*p0 := a2
*p1 := a3
```

```
__m128 _mm_movehl_ps (__m128 a, __m128 b) MOVHLPS
```

Moves the upper 2 SP FP values of *b* to the lower 2 SP FP values of the result. The upper 2 SP FP values of *a* are passed through to the result.

```
r3 := a3
r2 := a2
r1 := b3
r0 := b2
```

```
__m128 _mm_movelh_ps (__m128 a, __m128 b) MOVLHPS
```

Moves the lower 2 SP FP values of *b* to the upper 2 SP FP values of the result. The lower 2 SP FP values of *a* are passed through to the result.

```
r3 := b1
r2 := b0
r1 := a1
r0 := a0
```

```
__m128 _mm_loadl_pi(__m128 a, __m64 *p) MOVLPS reg, mem
```

Sets the lower two SP FP values with 64 bits of data loaded from the address *p*; the upper two values are passed through from *a*.

```
r0 := *p0
r1 := *p1
r2 := a2
r3 := a3
```

```
void _mm_storel_pi(__m64 *p, __m128 a)  MOVLPS mem, reg
```

Stores the lower two SP FP values of *a* to the address *p*.

```
*p0 := b0
*p1 := b1
```

```
int _mm_movemask_ps(__m128 a)                    MOVMSKPS
```

Creates a 4-bit mask from the most significant bits of the four SP FP values.

```
r := sign(a3)<<3 | sign(a2)<<2 | sign(a1)<<1 | sign(a0)
```

```
unsigned int _mm_getcsr(void)                    STMXCSR
```

Returns the contents of the control register.

```
void _mm_setcsr(unsigned int i)                  LDMXCSR
```

Sets the control register to the value specified.

Macro Function for Shuffle

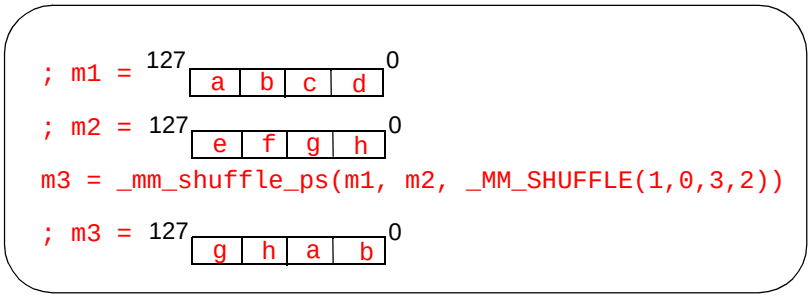
The Streaming SIMD Extensions provide a macro function to help create constants that describe shuffle operations. The macro takes four small integers (in the range of 0 to 3) and combines them into an 8-bit immediate value used by the SHUFPS instruction. See Example 13-1.

Example 13-1 Shuffle Function Macro

```
_MM_SHUFFLE(z,y,x,w)
  expands to the value of
  (z<<6) | (y<<4) | (x<<2) | w
```

You can view the four integers as selectors for choosing which two words from the first input operand and which two words from the second are to be put into the result word.

Example 13-2 View of Original and Result Words With Shuffle Function Macro



Macro Functions to Read and Write the Control Registers

The following macro functions enable you to read and write bits to and from the control register. For details, see "Set Operations" later in this chapter.

Exception State Macros

`_MM_SET_EXCEPTION_STATE(x)`

`_MM_GET_EXCEPTION_STATE()`

Macro Definitions

Write to and read from the sixth-least significant control register bit, respectively.

Macro Arguments
_MM_EXCEPT_INVALID
_MM_EXCEPT_DIV_ZERO
_MM_EXCEPT_DENORM
_MM_EXCEPT_OVERFLOW
_MM_EXCEPT_UNDERFLOW
_MM_EXCEPT_INEXACT

Example 13-3 tests for a divide-by-zero exception.

Example 13-3 Exception State Macros with _MM_EXCEPT_DIV_ZERO

```
if (_MM_GET_EXCEPTION_STATE(x) & _MM_EXCEPT_DIV_ZERO) {
    /* Exception has occurred */
}
```

Exception Mask Macros

`_MM_SET_EXCEPTION_MASK(x)`

Macro Arguments
_MM_MASK_INVALID

_MM_GET_EXCEPTION_MASK()	_MM_MASK_DIV_ZERO
Macro Definitions	_MM_MASK_DENORM
Write to and read from the seventh through twelfth control register bits, respectively. **Note:** *All six exception mask bits are always affected. Bits not set explicitly are cleared.*	_MM_MASK_OVERFLOW
	_MM_MASK_UNDERFLOW
	_MM_MASK_INEXACT

Example 13-4 masks the overflow and underflow exceptions and unmasks all other exceptions.

Example 13-4 Exception Mask with _MM_MASK_OVERFLOW and _MM_MASK_UNDERFLOW

```
_MM_SET_EXCEPTION_MASK( _MM_MASK_OVERFLOW | _MM_MASK_UNDERFLOW)
```

Rounding Mode	**Macro Arguments**
_MM_SET_ROUNDING_MODE(x)	_MM_ROUND_NEAREST
_MM_GET_ROUNDING_MODE()	_MM_ROUND_DOWN
	_MM_ROUND_UP
Macro Definition	_MM_ROUND_TOWARD_ZERO
Write to and read from bits thirteen and fourteen of the control register.	

Example 13-5 tests the rounding mode for round toward zero.

Example 13-5 Rounding Mode with _MM_ROUND_TOWARD_ZERO

```
if (_MM_GET_ROUNDING_MODE() == _MM_ROUND_TOWARD_ZERO) {
    /* Rounding mode is round toward zero */
}
```

Flush-to-Zero Mode	**Macro Arguments**
_MM_SET_FLUSH_ZERO_MODE(x)	_MM_FLUSH_ZERO_ON
_MM_GET_FLUSH_ZERO_MODE()	_MM_FLUSH_ZERO_OFF

Macro Definition
Write to and read from bit fifteen of the control register.

Example 13-6 disables flush-to-zero mode.

Example 13-6 Flush-to-Zero Mode with `_MM_FLUSH_ZERO_OFF`

```
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSHZERO_OFF)
```

Macro Function for Matrix Transposition

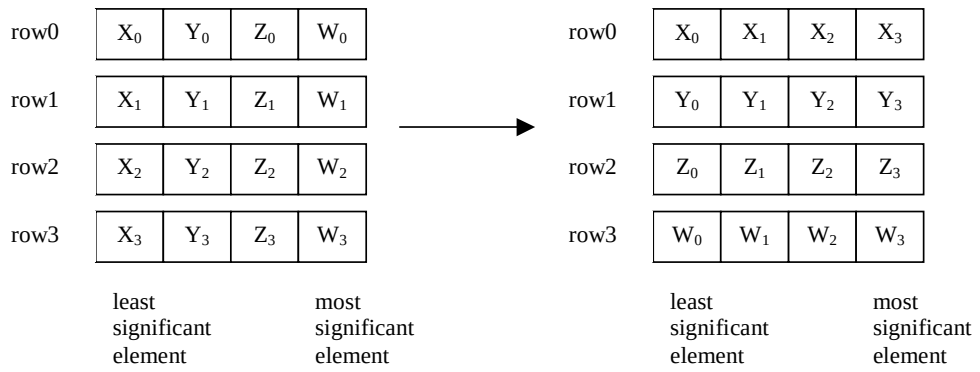
The Streaming SIMD Extensions also provide the following macro function to transpose a 4 by 4 matrix of single precision floating point values.

`_MM_TRANSPOSE4_PS(row0, row1, row2, row3)`

The arguments `row0`, `row1`, `row2`, and `row3` are `__m128` values whose elements form the corresponding rows of a 4 by 4 matrix. The matrix transposition is returned in arguments `row0`, `row1`, `row2`, and `row3` where `row0` now holds `column 0` of the original matrix, `row1` now holds `column 1` of the original matrix, and so on.

The transposition function of this macro is illustrated in Figure 13-2.

Figure 13-2 Matrix Transposition Using the `_MM_TRANSPOSE4_PS` Macro



Memory and Initialization

This section describes the Load, Set, and Store operations, which let you load and store data into memory. The Load and Set operations are similar in that both initialize `__m128` data. However, the Set operations take a float argument and are intended for initialization with constants, whereas the Load operations take a floating point argument and are intended to mimic the instructions for loading data from memory. The Store operation assigns the initialized data to the address.

Load Operations

`__m128 _mm_load_ss(float *p)` `MOVSS`

Loads an SP FP value into the low word and clears the upper three words.

```
r0 := *p
r1 := 0.0 ; r2 := 0.0 ; r3 := 0.0
```

`__m128 _mm_load1_ps(float *p)` `MOVSS` + shuffling

or

`__m128 _mm_load_ps1(float *p)` `MOVSS` + shuffling

Loads a single SP FP value, copying it into all four words.

```
r0 := *p
r1 := *p
r2 := *p
r3 := *p
```

`__m128 _mm_load_ps(float *p)` `MOVAPS`

Loads four SP FP values. The address must be 16-byte-aligned.

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

`__m128 _mm_loadu_ps(float *p)` `MOVUPS`

Loads four SP FP values. The address need not be 16-byte-aligned.

```
r0 := p[0]
r1 := p[1]
r2 := p[2]
r3 := p[3]
```

`__m128 _mm_loadr_ps(float *p)` `MOVAPS` + shuffling

Loads four SP FP values in reverse order. The address must be 16-byte-aligned.

```
r0 := p[3]
r1 := p[2]
r2 := p[1]
r3 := p[0]
```

Set Operations

`__m128 _mm_set_ss(float w)` (composite)

Sets the low word of an SP FP value to *w* and clears the upper three words.

```
r0 := w
r1 := r2 := r3 := 0.0
```

`__m128 _mm_set1_ps(float w)` (composite)

or

`__m128 _mm_set_ps1(float w)` (composite)

Sets the four SP FP values to *w*.

```
r0 := r1 := r2 := r3 := w
```

`__m128 _mm_set_ps(float z, float y, float x, float w)` (composite)

Sets the four SP FP values to the four inputs.

```
r0 := w
r1 := x
r2 := y
r3 := z
```

`__m128 _mm_setr_ps(float z, float y, float x, float w)` (composite)

Sets the four SP FP values to the four inputs in reverse order.

```
r0 := z
r1 := y
r2 := x
r3 := w
```

`__m128 _mm_setzero_ps(void)` (composite)

Clears the four SP FP values.

```
r0 := r1 := r2 := r3 := 0.0
```

Store Operations

`void _mm_store_ss(float *p, __m128 a)` MOVSS

Stores the lower SP FP value.

```
*p := a0
```

`void _mm_store1_ps(float *p, __m128 a)` MOVSS + shuffling

or

`void _mm_store_ps1(float *p, __m128 a)` MOVSS + shuffling

Stores the lower SP FP value across four words.

```
p[0] := a0
p[1] := a0
p[2] := a0
p[3] := a0
```

`void _mm_store_ps(float *p, __m128 a)` MOVAPS

Stores four SP FP values. The address must be 16-byte-aligned.

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

`void _mm_storeu_ps(float *p, __m128 a)` `MOVUPS`

Stores four SP FP values. The address need not be 16-byte-aligned.

```
p[0] := a0
p[1] := a1
p[2] := a2
p[3] := a3
```

`void _mm_storer_ps(float *p, __m128 a)` `MOVAPS` + shuffling

Stores four SP FP values in reverse order. The address must be 16-byte-aligned.

```
p[0] := a3
p[1] := a2
p[2] := a1
p[3] := a0
```

`__m128 _mm_move_ss(__m128 a, __m128 b)` `MOVSS`

Sets the low word to the SP FP value of *b*. The upper 3 SP FP values are passed through from a.

```
r0 := b0
r1 := a1
r2 := a2
r3 := a3
```

Integer Intrinsics

The integer intrinsics are listed in Table 13-7, followed by a description of each intrinsic with the most recent mnemonic naming convention. The alternate name is provided in case you have used these intrinsics before.

Table 13-7 Integer Intrinsics

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_extract_pi16	_m_pextrw	PEXTRW
_mm_insert_pi16	_m_pinsrw	PINSRW
_mm_max_pi16	_m_pmaxsw	PMAXSW
_mm_max_pu8	_m_pmaxub	PMAXUB

Table 13-7 Integer Intrinsics (Continued)

Intrinsic Name	Alternate Name	Corresponding Instruction
_mm_min_pi16	_m_pminsw	PMINSW
_mm_min_pu8	_m_pminub	PMINUB
_mm_movemask_pi8	_m_pmovmskb	PMOVMSKB
_mm_mulhi_pu16	_m_pmulhuw	PMULHUW
_mm_shuffle_pi16	_m_pshufw	PSHUFW
_mm_maskmove_si64	_m_maskmovq	MASKMOVQ
_mm_avg_pu8	_m_pavgb	PAVGB
_mm_avg_pu16	_m_pavgw	PAVGW
_mm_sad_pu8	_m_psadbw	PSADBW

For this section you need to ensure to empty the multimedia state for the mmx register. See "The EMMS Instruction: Why You Need it and When to Use it".

`int _mm_extract_pi16(__m64 a, int n)` PEXTRW

Extracts one of the four words of *a*. The selector *n* must be an immediate.

```
r := (n==0) ? a0 : ( (n==1) ? a1 : ( (n==2) ? a2 : a3 ) )
```

`__m64 _mm_insert_pi16(__m64 a, int d, int n)` PINSRW

Inserts word *d* into one of four words of *a*. The selector *n* must be an immediate.

```
r0 := (n==0) ? d : a0;
r1 := (n==1) ? d : a1;
r2 := (n==2) ? d : a2;
r3 := (n==3) ? d : a3;
```

`__m64 _mm_max_pi16(__m64 a, __m64 b)` PMAXSW

Computes the element-wise maximum of the words in *a* and *b*.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _mm_max_pu8(__m64 a, __m64 b)                PMAXUB
```

Computes the element-wise maximum of the unsigned bytes in *a* and *b*.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
__m64 _mm_min_pi16(__m64 a, __m64 b)               PMINSW
```

Computes the element-wise minimum of the words in *a* and *b*.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
r2 := min(a2, b2)
r3 := min(a3, b3)
```

```
__m64 _mm_min_pu8(__m64 a, __m64 b)                PMINUB
```

Computes the element-wise minimum of the unsigned bytes in *a* and *b*.

```
r0 := min(a0, b0)
r1 := min(a1, b1)
...
r7 := min(a7, b7)
```

```
int _mm_movemask_pi8(__m64 a)                      PMOVMSKB
```

Creates an 8-bit mask from the most significant bits of the bytes in *a*.

```
r := sign(a7)<<7 | sign(a6)<<6 | ... | sign(a0)
```

```
__m64 _mm_mulhi_pu16(__m64 a, __m64 b)             PMULHUW
```

Multiplies the unsigned words in *a* and *b*, returning the upper 16 bits of the 32-bit intermediate results.

```
r0 := hiword(a0 * b0)
r1 := hiword(a1 * b1)
r2 := hiword(a2 * b2)
r3 := hiword(a3 * b3)
```

```
__m64 _mm_shuffle_pi16(__m64 a, int n)                PSHUFW
```

Returns a combination of the four words of *a*. The selector *n* must be an immediate.

```
r0 := word (n&0x3) of a
r1 := word ((n>>2)&0x3) of a
r2 := word ((n>>4)&0x3) of a
r3 := word ((n>>6)&0x3) of a
```

```
void _mm_maskmove_si64(__m64 d, __m64 n, char *p)
MASKMOVQ
```

Conditionally store byte elements of *d* to address *p*. The high bit of each byte in the selector *n* determines whether the corresponding byte in *d* will be stored.

```
if (sign(n0)) p[0] := d0
if (sign(n1)) p[1] := d1
...
if (sign(n7)) p[7] := d7
```

```
__m64 _mm_avg_pu8(__m64 a, __m64 b)                   PAVGB
```

Computes the (rounded) averages of the unsigned bytes in a and b.

```
t = (unsigned short)a0 + (unsigned short)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned short)a7 + (unsigned short)b7
r7 = (unsigned char)((t >> 1) | (t & 0x01))
```

```
__m64 _mm_avg_pu16(__m64 a, __m64 b)                  PAVGW
```

Computes the (rounded) averages of the unsigned words in a and b.

```
t = (unsigned int)a0 + (unsigned int)b0
r0 = (t >> 1) | (t & 0x01)
...
t = (unsigned word)a7 + (unsigned word)b7
r7 = (unsigned short)((t >> 1) | (t & 0x01))
```

`__m64 _mm_sad_pu8(__m64 a, __m64 b)` PSADBW

Computes the sum of the absolute differences of the unsigned bytes in `a` and `b`, returning he value in the lower word. The upper three words are cleared.

```
r0 = abs(a0-b0) + ... + abs(a7-b7)
r1 = r2 = r3 = 0
```

Cacheability Support

The following intrinsics provide ways to make efficient use of the cache.

`void _mm_prefetch(char *p, int i)` PREFETCH

Loads one cache line of data from address *p* to a location "closer" to the processor. The value *i* specifies the type of prefetch operation: the constants `_MM_HINT_T0`, `_MM_HINT_T1`, `_MM_HINT_T2`, and `_MM_HINT_NTA` should be used, corresponding to the type of prefetch instruction.

`void _mm_stream_pi(__m64 *p, __m64 a)` MOVNTQ

Stores the data in *a* to the address *p* without polluting the caches. This intrinsic requires you to empty the multimedia state for the `mmx` register. See "The EMMS Instruction: Why You Need it and When to Use it".

`void _mm_stream_ps(float *p, __m128 a)` MOVNTPS

Stores the data in *a* to the address *p* without polluting the caches. The address must be 16-byte-aligned.

`void _mm_sfence(void)` SFENCE

Guarantees that every preceding store is globally visible before any subsequent store.

Data Alignment

To improve performance, you should align data to 16 bytes in memory operations when you are using the Streaming SIMD Extensions. Specifically, you must align `__m128` objects as addresses passed to the `_mm_load` and `_mm_store` intrinsics. If you want to declare arrays of floats and treat them as `__m128` objects by casting, you need to ensure that the float arrays are properly aligned.

Alignment Support

Use `__declspec(align)` to direct the compiler to align data more strictly than it otherwise does. For example, a data object of type `int` is allocated at a byte address which is a multiple of 4 by default (the size of an `int`). However, by using `__declspec(align)`, you can direct the compiler to instead use an address which is a multiple of 8, 16, or 32 with the following restrictions:

- 32-byte addresses must be statically allocated
- 16-byte addresses can be locally or statically allocated

You can use this data alignment support as an advantage in optimizing cache line usage. By clustering small objects that are commonly used together into a `struct`, and forcing the `struct` to be allocated at the beginning of a cache line, you can effectively guarantee that each object is loaded into the cache as soon as any one is accessed, resulting in a significant performance benefit.

The syntax of this extended-attribute is as follows:

`align(`*n*`)`

where *n* is an integral power of 2, less than or equal to 32. The value specified is the requested alignment.

NOTE. *If a value is specified that is less than the alignment of the affected data type, it has no effect. In other words, data is aligned to the maximum of its own alignment or the alignment specified with* `__declspec(align)`.

You can request alignments for individual variables, whether of static or automatic storage duration. (Global and static variables have static storage duration; local variables have automatic storage duration by default.) However, you cannot adjust the alignment of a parameter, nor a field of a `struct` or `class`. However, you can increase the alignment of a `struct` (or `union` or `class`), in which case every object of that type is affected.

As an example, suppose that a function uses local variables `i` and `j` as subscripts into a 2-dimensional array. They might be declared as follows:

```
int i, j;
```

These variables are commonly used together. But they can fall in different cache lines, which could be detrimental to performance. You can instead declare them as follows:

```
__declspec(align(8)) struct { int i, j; } sub;
```

The compiler now ensures that they are allocated in the same cache line. In C++, you can omit the `struct` variable name (written as *`sub`* in the above example). In C, however, it is required, and you must write references to `i` and `j` as *`sub`*`.i` and *`sub`*`.j`.

If you use many functions with such subscript pairs, it is more convenient to declare and use a struct type for them, as in the following example:

```
typedef struct __declspec(align(8)) { int i, j; } Sub;
```

By placing the `__declspec(align)` after the keyword `struct`, you are requesting the appropriate alignment for all objects of that type. Hhowever, that allocation of parameters is unaffected by `__declspec(align)`. If necessary, you can assign the value of a parameter to a local variable with the appropriate alignment.)

You can also force alignment of global variables, such as arrays:

```
__declspec(align(16)) float array[1000];
```

Dynamic Stack Frame Alignment

By default, the compiler targets functions that use 8- or 16-byte objects for dynamic alignment. It is up to the discretion of the compiler to decide whether dynamic stack alignment is needed on a function-per-function level (for stability of performace). This ensures that references to `double` and `__m64` to `double`, `__m64`, and `__m128` local variables align to addresses evenly divisible by 16. Parameters alignment is not affected.

Table 13-8 shows how -Qsfalign controls the stack frame alignment .

Table 13-8 Stack Frame Alignment Options

Option	Meaning
-Qsfalign8	Align stack for frames or functions with variables of 8 or 16 bytes where it is deemed appropriate by the compiler.
-Qsfalign16	Align stack for frames or functions with 16-byte variables where it is deemed appropriate by the compiler.
-Qsfalign-	Disable stack alignment for all functions.

If you are using aligned frames, you should not modify the EBX register in inlined assembly blocks because EBX is used to keep track of the argument block. You can modify EBX only if you save and restore EBX each time you use it. The Intel C/C++ Compiler uses the EBX register to control alignment of variables of these data types so using EBX, without preserving it, will cause unexpected program execution..

For additional information on the use of EBX in inline assembly code and other related issues, see the Application Note AP-833 *Data Alignment and Programming Issues with the Intel C/C++ Compiler* (Order #243872-001), and AP-589 Software Conventions for the Streaming SIMD Extensions (Order # 243873-001).

Assembly Language Support

The Intel C/C++ Compiler supports inlining of assembly blocks and generation of assembly files.

Inline Assembly

The compiler supports use of all the MMX and Streaming SIMD Extensions in inline assembly (__asm) blocks. The compiler also accepts the new syntax MMWORD PTR and XMMWORD PTR to refer to 64- and 128-bit data.

Generation of Assembly files

Use the `-S` compiler switch to produce an assembly listing. This is useful for hand-tuning compiler-generated code. The assembly files can be directly compiled using MASM version 6.12 with `iaxmm.inc` MASM include files. Generated `.asm` files contain an include directive for them when the MMX technology intrinsics or Streaming SIMD Extensions are used. For details on using inline assembly, see "Producing an Assembly Code Listing (-S)" in Chapter 6 .

Compiler Vectorization Support and Guidelines

14

This chapter provides guidelines, option descriptions, and examples for compiler vectorization with respect to the Intel C/C++ Compiler. The following list provides a summary of the chapter contents.

- A Quick reference of vectorization functionality and features
- Descriptions of compiler switches to control vectorization
- Descriptions of the C/C++ language features to control vectorization
- Discussion and general guidelines on vectorization levels:
  - automatic vectorization
  - vectorization with user intervention
  - no vectorization whatsoever
- Examples demonstrating typical vectorization issues and resolutions

Vectorizer Quick Reference

The Intel C/C++ Compiler vectorization can be summarized by the command line switches in Table 14-1:

Table 14-1 Vectorization Command-Line Switches

Option	Description	Default	Reference
`-Qvec`	Enables the vectorizer.	OFF	page 14-2
`-Qvec_alignment`	Controls the default alignment of vectorizable data.	OFF	page 14-2
`-Qvec_verbose`	Controls the vectorizer's diagnostic levels.	`n=1`	page 14-3

continued

Table 14-1 Vectorization Command-Line Switches (Continued)

Option	Description	Default	Reference
`-Qrestrict`	Enables pointer disambiguation with the `restrict` qualifier.	OFF	page 14-3
`-Qkscalar`	Performs all 32-bit floating point arithmetic using the Streaming SIMD Extensions instead of the default x87 instructions.	OFF	page 14-3
`-Qvec_emms[-]`	Controls the automation of `EMMS` instruction insertions to empty the `mmx` instruction registers.	ON	page 14-3
`-Qvec_no_arg_alias[-]`	Assumes on entry that procedure arguments are not aliased.	OFF	page 14-4
`-Qvec_no_alias[-]`	Assumes that no aliasing can occur between objects with different names.	OFF	page 14-4

Command-Line Switch Support

The following vectorizer switch options are available from the compiler command line. As with many advanced features of compilers, you must be sure to properly understand the functionality of the switches in order to use them effectively and avoid unwanted program behavior.

`-Qvec` Enables the vectorizer. All other vectorization switches require this switch to on. The default setting is off .

`-Qvec_alignment[n]`

Controls the default alignment of data that can be vectorized. By default, the compiler assumes a natural alignment; for example, a `float` array aligned to 4 bytes generates *unaligned* memory operations. However, if the compiler proves proper alignment, then it uses aligned instructions to improve performance. To override the compiler's default alignment, use the sub options as follows:

`n=0:` default alignment

`n=1:` assumes unaligned

`n=2:` assumes aligned

This feature is commonly used in programs that do not use `__declspec(align)` to pass data with pointers. See "Data Alignment" in Chapter 13.

CAUTION. *If you change the default alignment, you must be absolutely sure . Otherwise, the compiler will generate incorrect code.*

`-Qvec_emms[-]`

Controls whether the compiler inserts an `EMMS` instruction after each integer loop vectorized. See "The EMMS Instruction: Why You Need it and When to Use it" in Chapter 13.

CAUTION. *If you cannot tell whether your program uses MMX™ technology, only use* `-Qvec_emms[-]` *when you know it is safe to do so.*

`-Qrestrict` The `restrict` keyword is a qualifier that allows the disambiguator flexibility in alias assumptions, which enables more vectorization and other optimizations. Because the `restrict` keyword is a language extension, you must use `-Qrestrict` to enable it.

`-Qkscalar` Specifies 32-bit floating point arithmetic to operate on 32-bit scalar operations with Streaming SIMD Extensions.

`-Qvec_verbose[n]`

Controls the vectorizer's level of diagnostic messages:

- `n=0:` no diagnostic information is displayed
- `n=1:` display diagnostics indicating loops successfully vectorized (default)
- `n=2:` same as `n=1`, plus diagnostics indicating loops not successfully vectorized

`n=3:` same as `n=2`, plus additional information about any proven or assumed dependences

`-Qvec_no_arg_alias[-]`

Assumes that procedure arguments are not aliased on entry, neither by direct nor indirect referencing. The loop in Example 14-1 will not vectorize unless you use the `-Qvec_no_arg_alias` switch.

Example 14-1 Using `-Qvec_no_arg_alias` to Vectorize

```
void f(float *out, float *in){
  int i;
  for (i = 0; i < 100; i++)
    out[i] = in[i];
}
```

CAUTION. *If you incorrectly assert that procedure arguments are not aliased, you will generate improper program behavior. This caution* applies to `-Qvec_no_arg_alias` *and* `-Qvec_no_alias`.

`-Qvec_no_alias[-]`

Assumes that no aliasing occurs between objects with different names, neither by direct nor indirect referencing. The loop in Example 14-2 does not vectorize unless you use the `-Qvec_no_alias.`

Example 14-2 Using `-Qvec_no_alias` to Vectorize

```
void f(float *p){
int i;
float *out=p;
float *in=p;

for (i = 0; i < 100; i++)
  out[i] = in[i];
}
```

Language Support and Pragmas

This section addresses language features that help vectorize to the target hardware limitations and stylistic aspects. The `declspec(align(n))` declaration enables you to overcome hardware alignment constraints. The `restrict` qualifier and the pragmas address the stylistic issues due to lexical scope, data interdependence, and ambiguity resolution.

Table 14-2 Language Support

Option	Description	Reference
`declspec(align(n))`	Directs the compiler to align the variable `var-name` to an n-byte boundary.	page 14-6
`restrict`	Permits the disambiguator flexibility in alias assumptions, which enables more vectorization.	page 14-3
`#pragma ivdep`	Instructs the compiler to ignore assumed vector dependencies.	page 14-7
`#pragma vector {aligned | unaligned}`	Specifies how to vectorize the loop.	page 14-8
`#pragma novector`	Specifies that the loop should never be vectorized	page 14-9

Alignment with declspec

Syntax: `__declspec(align(n)) <type> var-name`

Definition: Directs the compiler to align the variable *var-name* to an *n*-byte boundary, where *n* must be between 2 and 32. See "Data Alignment" in Chapter 13.

Example 14-3 directs the compiler to align the `float` array to 16 bytes.

Example 14-3 Using declspec(align(n)) for Alignment

```
__declspec(align(16)) float x[1024];
```

Qualifying with the restrict Keyword

Syntax: `<type> *restrict p`

Definition: The `restrict` keyword is a qualifier. If `p` is a restricted pointer to an object `x`, then all references to `x` within the lexical scope of `p` must involve an expression with `p`. This allows the disambiguator flexibility in alias assumptions, which enables more vectorization and other optimizations. Typically, this feature is used to indicate that arguments to the function are not aliases.

Because the `restrict` keyword is a language extension, you must use `-Qrestrict` to enable it. Without the restricted pointers in the following, the compiler must assume that `a[i]` is aliased to `b[i-1]`, which prohibits vectorization.

Example 14-4 Using the `restrict` Keyword as a Qualifier

```
void foo (float *restrict a, float *b) {
  for (i=0; i<N; i++)
    a[i] = b[i] * 2.0f;
}
```

Pragma Scope

These pragmas control the vectorization of only the subsequent loop in the program, but the program does not apply them to any nested loops. Each nested loop needs its own pragma preceding it in order for the pragma to be applied. You must place a pragma only before the loop control statement. However, assignments and function calls can appear between the pragma and the loop it affects.

#pragma ivdep

Syntax: `#pragma ivdep`

Definition: This pragma instructs the compiler to ignore assumed vector dependences. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This pragma overrides that decision. Only use this when you know that the assumed loop dependences are safe to ignore.

The loop in Example 14-5 will not vectorize without the `ivdep` pragma, since `k` is not known (and it could be `k<0`).

Example 14-5 Loop Using #pragma ivdep

```
#pragma ivdep
  for (i=0; i<m; i++) {
    a[i] = a[i+k] * c;
...
  }
```

#pragma vector

Syntax: `#pragma vector {aligned |unaligned}`

Definition: The `vector` loop pragma means the loop should be vectorized, if it is legal to do so, ignoring normal heuristic decisions about profitability. When the `aligned` (or `unaligned`) qualifier is used with this pragma, the loop should be vectorized using `aligned` (or `unaligned`) operations. Specify one and only one of `aligned` or `unaligned`. Using pragma to control alignment in a loop overrides `-Qvec_alignment[n]`.

The loop in Example 14-6 uses the vector pragma to indicate that, despite the stride-2 references, the loop should be vectorized.

Example 14-6 Loop Using Stride-2 #pragma vector

```
#pragma vector
  for (i=0; i<m; i+=2) {
    a[i] = a[i] * c;
  }
```

This loop in Example 14-7 uses the aligned qualifier to request that the loop be vectorized with aligned instructions, as the arrays are declared in such a way that the compiler could not normally prove this would be safe to do so.

Example 14-7 Loop Using #pragma vector aligned

```
void f(float *a) {
#pragma vector aligned
  for (i=0; i<m; i+=2) {
    a[i] = a[i] * c;
  }
}
```

#pragma novector

Syntax: `#pragma novector`

Definition: The `novector` loop pragma specifies that the loop should never be vectorized, even if it is legal and profitable to do so.

In example Example 14-8, the `novector` pragma indicates that the programmer knows the tripcount is too low to make vectorization be worthwhile. This is indicated by the value of (`ub - lb)` being small.

Example 14-8 Low Trip Count Loop Using #pragma novector

```
void f(int lb, int ub) {
...
#pragma novector
  for (j=lb; j<ub; j++) {
  a[j] = a[j] + b[i];
  }
...
}
```

Loop Structure Coding Background

The goal of vectorizing compilers is to parallelize source code automatically. However, the realization of this goal has been difficult to achieve even for well-known vectorizing compilers. The reason for the difficulty in achieving automatic parallelization is due to two major factors:

1. *Style*—The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove two memory references are distinct locations. Consequently, this prevents certain reordering transformations.
2. *Hardware Restrictions*—The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to stride-1 accesses with a preference to 16-byte aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it to a distinct target architecture.

Many stylistic issues that prevent the automatic parallelization by vectorization compilers are found in loop structures.The ambiguity arises from the complexity of the keywords, operators, data references, and memory operations within the loop bodies.

However, by understanding these limitations and by knowing how to interpret diagnostic messages, you can modify your program to overcome the historical difficulty and enable optimal vectorizations. The following sections summarize the capabilities and restrictions of the vectorizer with respect to loop structures.

Key Programming Guidelines

Review these guidelines, restrictions, and examples, and check them against your code to eliminate ambiguities that prevent the compiler from achieving optimal vectorization.

- Guidelines for loop bodies:
  - Use straight-line code (a single basic block)
  - Use vector data only; that is, arrays, and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.

  - Use only assignment statements
- *Avoid* the following in loop bodies:
  - Function calls
  - Unvectorizable operations
  - Mixing vectorizable types in the same loop
  - Multiple types of SIMD operations
  - Use relational operators or the `?:` operator
  - Data-dependent loop exit conditions

Loop Constructs

Loops can be formed with the usual `for` and `while-do`, or repeat-until constructs or by using a `goto` or a label. However, the loops must have a single entry and a single exit to be vectorized.

Example 14-9 Loop Construct Usage

Correct Usage	Incorrect Usage

`for (i=0; i<n; i++) {`
`  ...`
`  }`
	`while (i<n) {`
	`  ...`
	`  if (cond) break;`
	`/* 2nd exit */`
	`  …`
	`}`

Loop Body Control Flow

Loop bodies must be straight-line code, that is, no branches.

Example 14-10 Loop Body Control Flow

Correct Usage:

```
do {
    /* body is a single basic block */
    a[i] = b[i] * x
    b[i] = c[i] + sqrt(d[i]);
    --count;
  } while (count != 0);
```

Inorrect Usage:

```
while (i<n) {
    /* if-branch inside body of loop */
    a[i] = b[i] * c[i];
    if (a[i] < 0.0) {
    a[i] = 0.0;
    }
}
```

Loop Exit Conditions

Loop exit conditions determine the number of iterations that a loop executes. For example, fixed indexes in `for` loops determine the iterations. The loop iterations must be countable; that is, the number of iterations must be expressed as one of the following:

- a constant
- a linear function of an integer variable
- a loop invariant terms

Loops whose exit depends on computation are not countable.

Example 14-11 Loop Usage Comparisons

Correct Usage for Countable Loop:

```
count = N; /* exit condition specified by "N - lb + 1" */
...
do {
a[i] = b[i] * x
b[i] = c[i] + sqrt(d[i]);
--count;
} while (count != lb); /* lb is not defined within loop */
```

Correct Usage for Countable Loop:

```
 /* exit condition is "(n-m+2)/2" */
    i = 0;
    for (l=m; l<n; l+=2) {
    a[i] = b[i] * x
    b[i] = c[i] + sqrt(d[i]);
    ++i;
    }
```

Incorrect Usage for Non-Countable Loop:

```
i = 0;
/* iterations dependent on a[i] */
while (a[i] > 0.0) {
   a[i] = b[i] * c[i];
   ++i;
}
```

Stripmining and Cleanup

The compiler will automatically stripmine your loop and generate a cleanup loop. This means you do not need to unroll your loops.

Example 14-12 Strip Mining and Cleanup Loops

```
i = 0;
while (i<n) {
   a[i] = b[i] + c[i]; /* Code Clean-up Loop. */
   ++i;
}
/* The vectorizer will generate the following two loops. */
i = 0;
while (i < (n - n%4)) {
a[i] = b[i] + c[i]; /* Vector Loop. */
++i;
}
while (i < n) {
   a[i] = b[i] + c[i];/* Scalar clean-up loop. */
   ++i;
}
```

Statements in the Loop Body

The vectorizable operations are different for floating point and integer data.

Floating-point array operations

The statements within the loop body may be `float` operations (typically on arrays). Arithmetic operations are limited to addition, subtraction, multiplication, division, and negation. Note that conversion to/from floats is not allowed. Operation on `double` types is not allowed.

Integer Array Operations

The statements within the loop body may be `char`, `unsigned char`, `short`, `unsigned short`, `int`, and `unsigned int` operations (again, typically for arrays). Arithmetic operations are limited to addition, subtraction, bitwise `AND`, `OR`, and `XOR` operators.

Other Integer Operations

You cannot mix data types except for integer induction variables.

Other Datatypes

No statements other than the preceding floating point and integer operations are allowed. In particular, note that the special `__m64` and `__m128` datatypes are not vectorizable.

No Function Calls

The loop body may not contain any function calls. The current version does not support vectorization of the math intrinsics `sqrt` and `fabs`. Use of the Streaming SIMD Extensions intrinsics (`_mm_add_ps`) are not allowed.

Vectorizable Data References

For any data reference, either as an array element or pointer reference, take care to ensure that there are no potential dependence or alias constraints preventing vectorization; intuitively, an expression in one iteration must not depend on the value computed in a previous iteration and pointer variables must provably point to distinct locations. Use of the `ivdep` pragma and the `restrict` keyword can be used to tell the compiler to ignore assumed dependences. See also the examples in the "Data alignment" section.

Arrays

Vectorizable data in a loop may be expressed as uses of array elements, provided that the array references are stride-1 (4 bytes) or loop invariant. Stride-1 references are not vectorized by default; the `vector` pragma can be used to override this.

Pointers Vectorizable data can also be expressed using pointers, subject to the same constraints as uses of array elements: the references that are stride-1 or loop invariant. Non-unit stride references for integers are not supported.

Invariants Vectorizable data can also include loop invariant references on the right hand inside an expression, either as variables or numeric constants. The loop in Example 14-13 will vectorize:

Example 14-13 Vectorizable Loop Invariant Reference

```
for (i=0; i<n; i++) {
  a[i] = b[i] * 3.14f + c[j];
}
```

Data alignment

If vectorizable `float` data is provably aligned, the compiler will generate aligned instructions. This is the case for locally declared data and data declared using the alignment `declspec`. Where data alignment is not known, unaligned references will be used unless a pragma or command-line switch is used to override this as described in "Alignment with declspec".

Common Errors in Making Code Vectorization-Compatible

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make *only* the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- Do not unroll your loops, the compiler does this automatically.
- Do not decompose one loop with several statements in the body into several single-statement loops.

- Do not manually insert calls to EMMS; for example, via the _m_empty intrinsic, after the loops to be vectorized, the compiler does this by default (use -Qvec_emms to override the default). See "-Qvec_emms[-]" section earlier in this chapter.

Some Examples

This section contains a few simple examples of some common issues in vector programming.

Argument Aliasing: A Vector Copy

The loop in Example 14-14, a vector copy operation, does not vectorize because the compiler cannot prove dest[i] and src[i] are distinct.

Example 14-14 Unvectorizable Copy Due to Unproven Distinction

```
void vec_copy(float *dest, float *src, int len){
    int i;
    for (i=0; i<len; i++)
      dest[i] = src[i];
}
```

The restrict keyword indicates that the pointers refer to distinct objects. Therefore, the compiler allows vectorization.

Example 14-15 Using restrict to Prove Vectorizable Distinction

```
void vec_copy(float restrict *dest, float restrict *src, int len){
   int i;
   for (i=0; i<len; i++)
       dest[i] = src[i];
}
```

Data Alignment: Two Examples

Data Alignment Examples

The next function in Example 14-16 has a loop that will vectorize but only with unaligned memory instructions because it uses global variables for which the compiler can neither prove nor force 16-byte alignment.

Example 14-16 Unaligned Loop Due to Global Variables

```
float z[N], a[N], y[N], x;
void f(void)
{
  for (i=0; i<N; i++) {
    a[i] = a[i] * x + y[i];
   }
}
```

Use the `declspec` alignment to improve the loop as in Example 14-17.

Example 14-17 Loop Alignment with declspec

```
__declspec(align(16)) float z[N], a[N], y[N], x;
void f(void)
{
  for (i=0; i<N; i++) {
    a[i] = a[i] * x + y[i];
   }
}
```

Example 14-18 has a loop that vectorizes but only with unaligned memory instructions. The compiler can align the local arrays, but because `lb` is not known at compile-time, the correct alignment cannot be determined.

Example 14-18 Loop Unaligned Due to Unknown Variable Value at Compile Time

```
void f(int lb){
  float z2[N], a2[N], y2[N], x2;
  ...
  for (i=lb; i<N; i++) {
    a2[i] = a2[i] * x2 + y2[i];
}
   ...
}
```

If you know that `lb` is a multiple of `4`, you can align the loop with `#pragma vector aligned` as shown in Example 14-19.

Example 14-19 Alignment Due to Assertion of Variable as Multiple of 4

```
void f(int lb)
{
  float z2[N], a2[N], y2[N], x2;
  ...
  assert(lb%4==0);
  #pragma vector aligned
  for (i=lb; i<N; i++) {
    a2[i] = a2[i] * x2 + y2[i];
  }
  ...
}
```

The use of the assertion checks that the constraint `lb` is a multiple of `4`.

Data Dependence

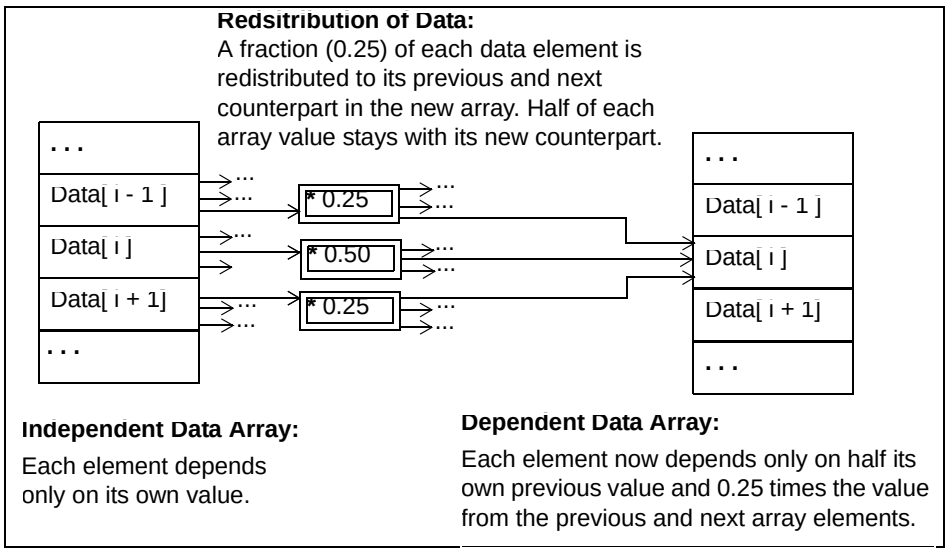
Example 14-20 shows some code that exhibits data dependence. Each element of an array is changed to be function of itself and its two neighbors.

Example 14-20 Data Dependent Loop

```
float data[N];
int i;
  for (i=1; i<N-1; i++) {
    data[i] = data[i-1]*0.25 + data[i]*0.5 + data[i+1]*0.25
}
```

Figure 14-1 shows how Example 14-20 exhibits data dependence.

Figure 14-1 A Redistribution for Data Dependence



This loop from Example 14-20 is not vectorizable because the write to the current element `data[i]` is dependent on the use of the preceding element `data[i-1]`, which has already been written to and changed in the previous iteration.

To see this, look at the access patterns of the array for the first two iterations as shown in Example 14-21

Example 14-21 Data Dependence Vectorization Patterns

Unvectorizable Access Pattern

```
i=1:READ data[0]
    READ data[1]
    READ data[2]
  WRITE data[1]

i=2:READ data[1]
    READ data[2]
    READ data[3]
  WRITE data[2]
```

Vectorizable Access Pattern

```
i=1,i=2:READ data[0]
        READ data[1]
        READ data[2]
        READ data[1]
        READ data[2]
        READ data[3]
        READ data[2]
      WRITE data[1]
      WRITE data[2]
```

In the normal sequential version of this loop shown in the Unvectorizable Access Pattern, the value of `data[1]` read from during the second iteration was written to in the first iteration.

For vectorization, the iterations must be done in parallel as in the Vectorizable Access Pattern. In this version, the value of `data[1]` read is the original value, not the updated version. You might be tempted to rewrite the loop in a vectorizable style by using a temporary array followed by a copy operation as shown in Figure 14-22.

Example 14-22 Data Independent but Still Unvectorizable Loop

```
float data[N], newdata[N];
int i;
  for (i=1; i<N-1; i++) {
    newdata[i] = data[i-1] * 0.25 + data[i]*0.5 + data[i+1]*0.25
  }
  for (i=1; i<N-1; i++) {
      data[i] = newdata[i];
  }
```

While both of these loops will vectorize, this computation *will produce a different result* than the original computation because you have changed the semantics of the original loop.

Loop Interchange and Subscripts: Matrix Multiply

Matrix multiplication is commonly written as shown in Example 14-23.

Example 14-23 Typical Matrix Multiplication

```
for (i=0; i<N; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

The use of `b[k][j]`, is not a stride-1 reference and therefore will not normally be vectorizable. The use of `c[i][j]` on the left-hand side also prevents vectorization; invariant references on the left-hand side are not allowed. If the loops are interchanged, however, all the references will become stride-1 as in Example 14-24.

Example 14-24 Matrix Multiplication with Stride-1

```
for (i=0; i<N; i++) {
  for (k=0; k<n; k++) {
    for (j=0; j<n; j++) {
c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

For Additional Information

The following sources might be useful in helping you understand basic vectorization terminology and technology:

- *High Performance Computing (2nd edition)*, Kevin Dowd (O'Reilly and Associates, 1998), ISBN 156592312X
- *Intel Architecture Optimization Manual*, Intel Corporation, order number, 730795.

Compiler Limits

A

Table A-1 shows the size or number of each item that the compiler can process. All capacities shown in the table are tested values; the actual number can be greater than the number shown.

Table A-1 Compiler Limits

Item	Tested Values
Control structure nesting (block nesting)	512
Conditional compilation nesting	512
Declarator modifiers	512
Parenthesis nesting levels	512
Significant characters, internal identifier	2048
External identifier name length	64K
Number of external identifiers/file	128K
Number of identifiers in a single block	2048
Number of macros simultaneously defined	128K
Number of parameters to a function call	512
Number of parameters per macro	512
Number of characters in a string	128K
Bytes in an object	512K
Include file nesting depth	512
Case labels in a switch	32K
Members in one structure or union	32K
Enumeration constants in one enumeration	8192
Levels of structure nesting	320

Experimental Performance Tuning

B

This appendix provides information on how you can adjust the compiler's optimization for a particular application by experimenting with interprocedural optimizations.

NOTE. *The suboptions described in this chapter are provided for the benefit of developers who are willing to experiment with advanced optimizations. Please be aware that experimenting with these advanced optimization can yield unexpected results. Therefore, this guide describes only the basic functionality of these advanced optimizations.*

Keywords for Optimization (-Qoption,*c*,*optlist*)

Enter the `-Qoption` option with the applicable keywords to select particular in-line expansions and loop optimizations. The option must be entered with a `-Qip` specification, as follows:

`-Qip [-Qoption,`*`c`*`,`*`optlist`*`]`

`c` is the specification that indicates the C compiler tool.

`optlist` is any applicable optimization for the specified level.

For example, the following disables the passing of arguments in registers:

`prompt>` **`icl -Qip -Qoption,c,ip_args_in_regs=0 a.cpp`**

The next section defines keywords to use with `-Qip` and `-Qoption.`

Interprocedural Optimization

If you specify `-Qip` without the `-Qoption` qualification, the compiler expands functions in line, propagates constant arguments, passes arguments in registers, and monitors module-level static variables. Use the following `-Qoption` keywords to refine these interprocedural optimizations:

`ip_args_in_regs=0`	Disables the passing of arguments in registers. By default, external functions can pass arguments in registers when called locally. Normally, only static functions can pass arguments in registers, provided the address of the function is not taken and the function does not use a variable number of arguments.
`ip_inline_max_calls=`*`N`*	Used with the inline heuristics (`-Qinl_heur`) option. This option changes the default number of call-sites to inline. Note that *`N`* call-sites are inlined only if that many call-sites meet the minimum inline criteria. For more information, see the "Using In-line Heuristics (-Qinl_heur n)" and "Criteria for In-Line Function Expansion" sections.
`ip_inline_max_stats=`*`n`*	Sets the allowable number of intermediate language statements and expressions for a function that is expanded in line. The number *`n`* is a positive integer. The number of intermediate language statements usually exceeds the actual number of source language statements. The default is set to the maximum number of 230.

`ip_inline_max_total_stats=`*n*	Sets the maximum increase in the number of intermediate language statements and expressions for each function that is expanded in line. The number *n* is a positive integer. By default, each function can increase to a maximum of 2000 statements.

Analyzing the Effects of Multifile IPO (-Qipo_c, -Qipo_S)

The `-Qipo_c` and `-Qipo_S` options are useful for analyzing the effects of multifile IPO, or when experimenting with multifile IPO between modules that do not make up a complete program.

Use the `-Qipo_c` option to optimize across files and produce an object file. This option performs optimizations as described for `-Qipo`, but stops prior to the final link stage, leaving an optimized object file. The default name for this file is `ipo_out.obj`. You can use the `-Fe` option to specify a different name as shown in the following example:

```
prompt> icl -G6 -Qipo_c -Fefilename a.cpp b.cpp c.cpp
```

Use the `-Qipo_S` option to optimize across files and produce an assembly file. This option performs optimizations as described for `-Qipo`, but stops prior to the final link stage, leaving an optimized assembly file. The default name for this file is `ipo_out.asm`. You can use the `-Fe` option to specify a different name as shown in the following example:

```
prompt> icl -G6 -Qipo_S -Fefilename a.cpp b.cpp c.cpp
```

Using In-line Heuristics (-Qinl_heur *n*)

Use the `-Qinl_heur` *n* option to inline the most frequently used call-sites. When you specify this option, call-sites are treated as leaf routines. For more information on in-lining and the minimum in-lining criteria, see "Controlling Inline Expansion of User Functions (-Obn, -Qip_no_inlining)" in Chapter 5.

The `-Qinl_heur n` option only takes effect when used together with the `-Qip` or `-Qipo` option. The value of *n* can be `0`, `1`, `2`, or `3`.

NOTE. *Inline heuristics `1` through `3` are all profile-guided. If no profile information is available the compiler uses the default inline heuristics (that is, `n = 0`) and issue a warning. See "Profile-Guided Optimization (PGO): Three Phases" in Chapter 5 for profile information procedures.*

There are four variations on the `-Qinl_heur n` option:

`-Qinl_heur 0`	Default in-line heuristics. These heuristics work both with or without available profile information.
`-Qinl_heur 1`	In-line the *N* most frequently executed call-sites.
`-Qinl_heur 2`	In-line the *N* most frequently executed call-sites. After each in-lining, the execution counts are scaled and the list of frequently executed call-sites is updated.
`-Qinl_heur 3`	In-line the *N* most frequently executed call-sites scaled by size of the callee. After each in-lining, the execution counts are scaled and the list of call-sites is updated.

By default, *N* = 100. You can change this value by using the option `-Qoption,c,-ip_inline_max_calls=N`. Note that *N* call-sites will be inlined only if that many call-sites meet the minimum inline criteria.

Criteria for In-Line Function Expansion

For a routine to be considered for in-lining, it has to meet certain minimum criteria. There are criteria to be met by the call-site, the caller, and the callee. The call-site is the site of the call to the function that might be in-lined. The caller is the function that contains the call-site. The callee is the function being called that might be in-lined.

Minimum call-site criteria:

- The number of actual arguments must match the number of formal arguments of the callee.

- The number of return values must be the same as the callees' number.
- The data types of the actual and formal arguments must be compatible.
- No multi-lingual in-lining is allowed. Caller and callee must be written in the same source language.

Minimum criteria for the caller:

- At most 2000 intermediate statements will be inlined into the caller from all the call-sites being inlined into the caller. You can change this value by specifying the option `-Qoption,c,-ip_inline_max_total_stats=` *`new value`*
- The function must be called or have its address used if it is declared as `static`. Otherwise, it will be deleted.

Minimum criteria for the callee:

- Is not considered infrequent due to the name. Routines that contain the following substrings in their names are not inlined: `abort`, `alloca`, `denied`, `err`, `exit`, `fail`, `fatal`, `fault`, `halt`, `init`, `interrupt`, `invalid`, `quit`, `rare`, `stop`, `timeout`, `trace`, `trap`, and `warn`.

Once these criteria are met, the compiler picks the routines whose inline expansions provide the greatest benefit to program performance. This is done using the following default heuristics. When you use profile-guided optimizations, a number of other heuristics are used (see "Profile-Guided Optimization (PGO): Three Phases" in Chapter 5 and "Using In-line Heuristics (-Qinl_heur n)" earlier in this chapter for more information).

- The default heuristic focuses on call-sites in loops or calls to functions containing loops.
- When profile information is available, the focus changes to the most frequently executed call-sites. Also, the default inline heuristic does not allow the in-lining of functions with more than 230 intermediate statements, or the number specified by the option `-Qoption,c,-ip_inline_max_stats`.
- The default inline heuristic stops when it detects direct recursion.
- The default heuristic will always inline very small functions that meet the minimum inline criteria. By default, functions with 15 or fewer intermediate statements are inlined. This limit can be modified with the option `-Qoption,c,-ip_inline_min_stats`.

Index

Symbols

A

B

C

D

E

F

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z